Stronger Consistency and Semantics for Low-Latency Geo-Replicated Storage

Wyatt Andrew Lloyd

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Michael J. Freedman

June 2013

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **JUN 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**
4. TITLE AND SUBTITLE **Stronger Consistency and Semantics for Low-Latency Geo-Replicated Storage**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Princeton University,Princeton,NJ,08544**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

Geo-replicated, distributed data stores that support complex online applications, such as social networks, strive to provide an always-on experience where operations always successfully complete with low latency. Today?s systems sacrifice strong consistency and expressive semantics to achieve these goals, exposing inconsistencies to their clients and necessitating complex application logic. In this work, we identify and define a consistency model?causal consistency with convergent conflict handling, or causal+?that is the strongest achieved under these constraints. We explore the ramifications of and implementation techniques for causal+ consistency through the design and implementation of two storage systems, COPS and Eiger. COPS is a key-value store that provides causal+ consistency. A key contribution of COPS is its scalability, which can enforce causal dependencies between keys stored across an entire cluster, rather than a single server like previous systems. Eiger is a system that provides causal+ consistency for the rich column-family data model. The central approach in COPS and Eiger is tracking and explicitly checking that causal dependencies between keys are satisfied in the local cluster before exposing writes. Our work also provides stronger semantics, in the form of read- and write-only transactions that allow programmers to consistently interact with data spread across many servers. COPS includes the first scalable read-only transactions that can optimistically complete in one round of local reads and take at most two non-blocking rounds of local reads. Eiger includes read-only transactions with the same properties that are more general?they are applicable to non-causal systems?and more fault-tolerant. Eiger also includes scalable write-only transactions that take three rounds of local non-blocking communication and do not block concurrent read-only transactions. We implemented COPS and Eiger and demonstrate their performance experimentally. The later of these systems, Eiger, achieves low latency, has throughput competitive with an eventually-consistent and non-transactional production system, and scales to large clusters almost linearly.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT **Same as Report (SAR)**	18. NUMBER OF PAGES **110**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Abstract

Geo-replicated, distributed data stores that support complex online applications, such as social networks, strive to provide an always-on experience where operations always successfully complete with low latency. Today's systems sacrifice strong consistency and expressive semantics to achieve these goals, exposing inconsistencies to their clients and necessitating complex application logic. In this work, we identify and define a consistency model—causal consistency with convergent conflict handling, or *causal+*—that is the strongest achieved under these constraints.

We explore the ramifications of and implementation techniques for causal+ consistency through the design and implementation of two storage systems, COPS and Eiger. COPS is a key-value store that provides causal+ consistency. A key contribution of COPS is its scalability, which can enforce causal dependencies between keys stored across an entire cluster, rather than a single server like previous systems. Eiger is a system that provides causal+ consistency for the rich column-family data model. The central approach in COPS and Eiger is tracking and explicitly checking that causal dependencies between keys are satisfied in the local cluster before exposing writes.

Our work also provides stronger semantics, in the form of read- and write-only transactions, that allow programmers to consistently interact with data spread across many servers. COPS includes the first scalable read-only transactions that can optimistically complete in one round of local reads and take at most two non-blocking rounds of local reads. Eiger includes read-only transactions with the same properties that are more general—they are applicable to non-causal systems—and more fault-tolerant. Eiger also includes scalable write-only transactions that take three rounds of local non-blocking communication and do not block concurrent read-only transactions.

We implemented COPS and Eiger and demonstrate their performance experimentally. The later of these systems, Eiger, achieves low latency, has throughput competitive with an eventually-consistent and non-transactional production system, and scales to large clusters almost linearly.

Acknowledgements

I had the opportunity to start my graduate studies concurrently with my adviser, Michael Freedman, starting on the tenure track. Our journey began with designing systems as we strolled among Princeton's neogothic buildings. Along the way we worked very hard, staying up all night to meet deadlines and spending a memorable week in the dim confines of the systems lab writing up COPS to submit to SOSP. He insisted upon finding problems grounded in reality. And on developing truly novel solutions to them. My taste in research will always bear his influence.

My thesis research was done in collaboration with my adviser, David Andersen, and Michael Kaminsky. David and Michael also served as unofficial secondary advisers for my latter three years in graduate school. David's Socratic style forced me to reexamine my assumptions and deepen my understanding of a subject. Michael's attention to detail and insistence on fully grasping every topic drove me to uncover corner cases and strengthened our designs. I would also like to thank the other members of my dissertation committee—Vivek Pai, Margaret Martonosi, and Jennifer Rexford—for their advice.

Throughout my time at Penn State and Princeton I had the opportunity to learn from many excellent faculty and peers. At Penn State, my passion for research was discovered with the help of faculty members John Hannan, Trent Jaeger, Guohong Cao, and Tom La Porta. Tom, my undergraduate thesis adviser, deserves special thanks for setting me on the path to graduate school. I also learned much from the graduate students I interacted with: Todd Arnold, Jing Zhao, Patrick Traynor, and William Enck.

At Princeton I found another vibrant intellectual community created by the senior systems faculty. I owe a particular debt to Jennifer Rexford who was always generous with her time and thorough in her outside review of my work. I would also like to thank the students and post-docs at Princeton who served as sounding boards and mentors: Siddhartha Sen, Jeff Terrace, David Shue, Muneeb Ali, Anirudh Badam, Steven Ko, Nate Foster, Erik Nordström, Changhoon Kim, and Yi Wang.

The research in this dissertation was supported by the National Science Foundation under a CNS ANET Award (#0831374), the Office of Naval Research (Young Investigator Award), and the Intel Science & Technology Center for Cloud Computing (ISTC-CC).

I would finally like to thank my family for their support, my grandparents Dot, Ed, Jean, and Dick, my parents Isabel and Errol, and my future wife Kristine.

To my future wife Kristine,

who gives depth to my life

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large-scale data stores are a critical infrastructure component in many of today's largest Internet services. These services support many millions of concurrent users interacting with many petabytes of data. The extreme scale of the infrastructure needed to support these services provides a new environment to explore distributed storage designs. In contrast to classical distributed storage systems that focus on local-area operation in the small, these services are typically characterized by massive wide-area deployments across a few to tens of datacenters, as illustrated in Figure 1.1.

In these systems each datacenter stores a full replica of the data, i.e., there is a copy of each data item in each datacenter.[1] For example, Facebook stores all user profiles, comments, friends lists, and likes at each of its datacenters [30]. Replicating data in this way across geographically distinct locations is called *geo-replication*. Geo-replication brings two key benefits to web services: fault tolerance and low latency. It provides fault tolerance through redundancy: if one location fails, other locations can continue to provide the service. It provides low latency through proximity: users can be directed to and served by a nearby location to avoid speed-of-light delays associated with cross-country or round-the-globe communication.

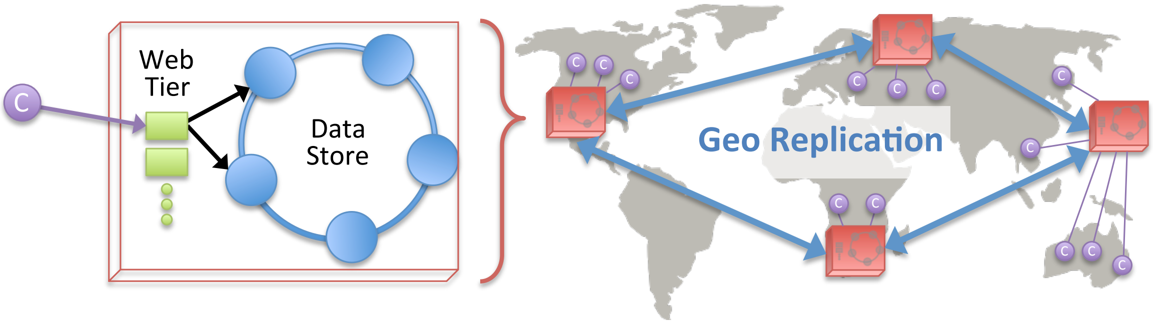


Figure 1.1: The general architecture of modern web services. Multiple geographically distributed datacenters each have application clients in the web tier that read and write state from a data store that is distributed across the datacenters, each of which distributes data across a large number of servers.

The scale of the data stored at each datacenter is far too large to fit on a single physical machine. Instead, it must be spread across clusters of 10s to 1000s of machines. A typical technique for spreading the data is *sharding*, which puts disjoint subsets of data on different servers. As a primitive example, Server 1 might store and serve user profiles for people whose names start with 'A', Server 2 for 'B', and so on. Such sharding allows clusters to scale in capacity and throughput. Adding additional machines to a cluster results in a rebalancing of shards across the new larger set of servers. Each server in the new cluster is then responsible for a smaller shard of the data. This allows the throughput and capacity of a cluster to increase, or *scale*, with the addition of more machines. The storage systems that support these large web services are thus doubly distributed systems. Replicas are distributed geographically and each is itself distributed across many machines via sharding.

When users initiate a connection with a service they are directed to a nearby datacenter. Within that datacenter users are a connected to one of many identical front-end web servers. Front-ends serve requests by reading and writing data to and from storage tier nodes. (In a three-tier web service the web server and application logic are considered separate tiers. We describe a two-tier web service where they are combined in this dissertation, but all discussion applies in the three-tier model as well with the application logic tier as our client.)

[1] Extending this work to settings where data may be only replicated over some subset of the datacenters is an interesting avenue of future work.

	New York City	**Los Angeles**	**Paris**	**Tokyo**	**Sydney**
Princeton to	8ms	72ms	110ms	195ms	240ms

Table 1.1: Example round trip times from Princeton to servers in the named cities.

For the storage system, the *clients* are the front-end web servers that issue read and write operations on behalf of the human users. When we say, “a client writes a value,” we mean that an application running on a web or application server writes into the storage system.

In this dissertation, we address the problem of building a geo-replicated data store targeted at applications that demand low latency. Such applications are common: Amazon, EBay, and Google all claim that a slight increase in user-perceived latency translates into concrete revenue loss [28, 29, 51, 65]. User-perceived latency consists of many components. These include, at a minimum, the network delay to the service instance a user is connected to; processing time on the front-end web server that accepts the user’s request; one or more rounds of accesses to the storage system; additional processing time on the front-end web server; the network delay for the response to travel back to the user; and user-side rendering time. Performance goals vary across services, but typical targets are in the hundreds or low thousands of milliseconds.

These low page-load-time targets help demonstrate the latency benefit of geo-replication: when less time is used for user-to-service-and-back network delay (also called round trip time or RTT) it easier to meet these targets. Table 1.1 shows some examples RTTs from a user in Princeton to servers around the world. Clearly, the page load time for a Princeton user will be much faster when accessing a service instance in New York City (8ms RTT) than when accessing one in Los Angeles (72ms RTT) or Sydney (240ms RTT).

Given the low targets for page load time and the many components that add to it, it is imperative that accesses to the data store have low latency. To this end, many storage systems—including those described in this dissertation—serve all accesses from the local replica of the data store. Reads return a current view of data from this replica; they do not examine remote replicas. Writes are applied at this replica immediately; they return before being applied at remote replicas.

The local-replica-only design helps ensure low latency by avoiding slow accesses to remote replicas, which would take at least the RTT between datacenters. Table 1.1 is again a good reference point for the magnitude of these RTTs that are avoided. The local-replica-only design, however, precludes providing the strongest forms of *consistency*, which restricts the execution of operations to make the data store more intuitive for users and programmers. For instance, linearizability, a strong consistency model, specifies that a read issued after a write returns must see the effects of that write. Linearizability is clearly incompatible with the local-replica-only design because writes return after being written to their local replicas, but a read issued to a different replica instantaneously after that write returns must see its effects even though it is impossible for them to have propagated to the other replicas yet due to the speed-of-light delay. Theoretical results formalize the trade-off between low latency and the strongest forms of consistency [10, 52].

The strongest forms of consistency are also at odds with other desirable properties of geo-replicated storage systems. The famous CAP theorem, conjectured by Brewer [17] and proved by Gilbert and Lynch [37], shows it is impossible to create a system that is linearizable, always available for reads and writes, and able to continue operating during network partitions. As a result, many modern web services have chosen to embrace availability and partition tolerance at the cost of linearizability [27, 35, 46]. We refer to scalable systems with these three properties as *ALPS* systems, because they provide **A**vailability, low **L**atency, **P**artition-tolerance, and high **S**calability. We define the ALPS properties more rigorously and give an in-depth discussion of the trade-offs with consistency in Chapter 2.

Given that ALPS systems must sacrifice strong consistency (i.e., linearizability), we seek the strongest consistency model that is achievable under these constraints. Such "stronger" consistency is desirable because it makes systems easier for a programmer to reason about and exposes fewer anomalies to end users of the system. In this paper, we consider *causal* *consistency with convergent conflict handling*, which we refer to as *causal+ consistency*. Many previous systems believed to implement the weaker causal consistency, such as Bayou [59] and PRACTI [12] actually implement the more useful causal+ consistency, though none do so in a scalable manner.

The causal component of causal+ consistency ensures that the data store respects the causal dependencies between operations [47]. Consider a scenario where a user uploads a picture to a web site, the picture is saved, and then a reference to it is added to that user's album. The reference "depends on" the picture being saved. Under causal+ consistency, these dependencies are always satisfied. Programmers never need to reason about the situation where they can get the reference to the picture but not the picture itself, unlike in systems with weaker guarantees, such as eventual consistency.

The convergent conflict handling component of causal+ consistency ensures that replicas never permanently diverge and that conflicting updates to the same key are dealt with identically at all sites. When combined with causal consistency, this property ensures that clients see only progressively newer versions of keys. In comparison, eventually consistent systems may expose versions out of order. By combining causal consistency and convergent conflict handling, causal+ consistency ensures clients see a causally-correct, conflict-free, and always-progressing data store.

Beyond causal+ consistency, we seek *stronger semantics* from our data store that give programmers more powerful tools for interacting with the data. These stronger semantics include a rich data model, read-only transactions, and write-only transactions.

Key-value storage—perhaps the simplest data model provided by data stores—is used today by many services [5, 34]. This data model, however, does not provide the abstractions typically used to build web services such as counters, lists, and graphs. In contrast, the column-family data models offered by systems like BigTable [23] and Cassandra [46] do provide these abstractions. These rich data models provide hierarchical sorted column-families and numerical counters. Column-families are well matched to many services, e.g., Facebook, while counter columns are particularly useful for numerical statistics, as used by collaborative filtering (Digg, Reddit), likes (Facebook), or re-tweets (Twitter). In this work, we demonstrate how to provide causal+ consistency for the simple key-value data model and the rich column-family data model.

In a scalable setting where data is spread across many machines, asynchronous requests to a consistently-updated data store are insufficient for providing consistency to clients of the system. Simultaneously issued requests to different servers may arrive at different times, thus returning distinct and potential inconsistent views of the data. Instead, what is needed are *read-only transactions* that provide programmers with a consistent and up-to-date view of data spread across many machines in the system. *Write-only transaction* similarly benefit programmers by enabling them to atomically update data spread across many machines in the system.

In this dissertation we describe three systems that provide causal+ consistency: COPS, COPS-RT, and Eiger. Each progressing system is a step forward for scalable geo-replicated storage, with Eiger superseding our original efforts on COPS and COPS-RT.

Our COPS system (Clusters of Order-Preserving Servers) provides causal+ consistency and is designed to support complex online applications that are hosted from a small number of large-scale datacenters, each of which is composed of front-end servers (clients of COPS) and back-end key-value data stores. COPS executes all read and write operations in the local datacenter in a linearizable fashion, and then replicates data across datacenters in a causal+ consistent order in the background.

COPS-RT[2] builds on COPS by adding *read-only transactions* that give clients a consistent view of multiple keys spread across many servers. Read-only transactions are needed to obtain a consistent view of multiple keys, even in a fully-linearizable system. Our read-only transactions require no locks, are non-blocking, and take at most two parallel rounds of intra-datacenter requests. These read-only transactions do come at some cost: compared to the

[2] The RT in COPS-RT stands for "read-only transactions." In the original COPS paper [53] we called these "get transactions" and referred to COPS-GT instead of COPS-RT.

regular version of COPS, COPS-RT is less efficient for certain workloads (e.g., write-heavy) and is less robust to long network partitions and transient datacenter failures.

Our Eiger system continues this line of work by providing stronger semantics for ALPS systems. These stronger semantics include providing causal+ consistency for the rich column-family data model, improved read-only transactions, and write-only transactions. Eiger's read-only and write-only transaction algorithms each represent an advance in the state-of-the-art. COPS introduces a read-only transaction algorithm that normally completes in one round of local reads, and two rounds in the worst case. Eiger's read-only transaction algorithm has the same properties, but achieves them using logical time instead of explicit dependencies. Not storing explicit dependencies allows Eiger to avoid the drawbacks of COPS-RT; Eiger is as efficient as COPS for all workloads and is robust to long network partitions and transient datacenter failures.

The scalability requirements for ALPS systems creates the largest distinction between this work and prior causal+ consistent systems. Previous systems required that all data fit on a single machine [2, 15, 59] or that all data that potentially could be accessed together fit on a single machine [12]. In comparison, data stored in COPS and Eiger can be spread across an arbitrarily-sized datacenter, with dependencies, read-only, and write-only transaction that can stretch across many servers in the datacenter. To the best of our knowledge, our work describes the first scalable systems to implement causal+ (and thus causal) consistency.

The contributions of this dissertation are as follows:

- We explicitly identify four important properties of distributed data stores and use them to define ALPS systems.
- We name and formally define causal+ consistency.
- We present the design and implementation of COPS, the first scalable system that efficiently realizes the causal+ consistency model.

- We present the design and implementation of Eiger, a causally-consistent data store based on a column-family data model, including all the intricacies necessary to offer abstractions such as column families and counter columns.

- We present a non-blocking, lock-free read-only transaction algorithm in COPS-RT that provides clients with a consistent view of multiple keys in at most two rounds of local operations.

- We present an improved non-blocking, lock-free read-only transaction algorithm in Eiger that is performant and partition tolerant.

- We present a novel write-only transaction algorithm that atomically writes a set of keys, is lock-free (low latency), and does not block concurrent read transactions.

- We show through evaluation that COPS has low latency, high throughput, and scales well for all tested workloads; and that COPS-RT has similar properties for common workloads.

- We show through evaluation that Eiger has performance competitive to eventually-consistent, non-transactional Cassandra.

Chapter 2

ALPS Systems and Trade-offs

Distributed storage systems have multiple, sometimes competing, goals: *availability*, *low latency*, and *partition tolerance* to provide an "always on" user experience [27]; *scalability* to adapt to increasing load and storage demands; and a sufficiently strong *consistency* model to simplify programming and provide users with the system behavior that they expect. In slightly more depth, the desirable properties include:

1. Availability. All operations issued to the data store complete successfully. No operation can block indefinitely or return an error signifying that data is unavailable.

Availability is important for data storage because it ensures clients can interact with a functioning service. Unresponsive storage results in an unresponsive service that quickly becomes an unused service. Our definition of availability also importantly avoids designs that "cheat" to provide low latency by quickly returning an error.

2. Low Latency. Client operations complete "quickly." Commercial service-level objectives suggest average performance of a few milliseconds and worst-case performance (i.e., 99.9th percentile) of 10s or 100s of milliseconds [27].

Low latency is critical for data storage as discussed in Chapter 1: it prevents slow page load times that are correlated with lost revenue.

3. Partition Tolerance. The data store continues to operate under network partitions, e.g., one separating datacenters in Asia from the United States. The failure of a datacenter is equivalent to it being partitioned from the rest of the system, and thus a partition tolerant system also continues to operate during the failure of a datacenter.

Partition tolerance and availability are different, but complementary requirements. Systems can provide one property but not the other. Quorum-based systems are considered partition tolerant because the "majority" partition can continue operating, but are not considered available because nodes in the "minority" partition cannot form a valid quorum and thus are unable to complete operations [17]. A system that assumes away partitions could be describes as available, even though that property would not hold in practice if a partition did occur.

Providing both partition tolerance and availability guarantees that all replicas of a system will continue to operate and be available as long as they have not failed.

4. High Scalability. The data store scales linearly. Adding N resources to the system increases aggregate throughput and storage capacity by $\mathrm{O}(N)$.

Scalability is paramount for modern storage systems. The amount of data backing web services continues to grow. Systems with all data fitting on a single machine are not longer practical. Instead, scalable storage must be used that spreads data across many machines and that can grow along with the data.

Enabling larger storage systems enables supporting a larger number of users, which is an important goal for almost all web services. Larger storage also enables an increase in the amount of storage per user that can be used to provide a richer web service, which is also an important goal for many web services.

5. Stronger Consistency. The data store provides consistency stronger than eventual consistency, which we define in this dissertation as achieving causal+ consistency. We define causal+ consistency in Chapter 3.

An ideal data store would provide *linearizability*—sometimes informally called *strong consistency*—which dictates that operations appear to take effect across the entire system at a single instance in time between the invocation and completion of the operation [41]. In a data store that provides linearizability, as soon as a client completes a write operation to an object in one datacenter, read operations to the same object in all other datacenters will reflect its newly written state. Linearizability simplifies programming—the distributed system provides a single, consistent image—and users experience the storage behavior they expect. Weaker, eventual consistency models, common in many large distributed systems, are less intuitive: not only might subsequent reads not reflect the latest value, reads across multiple objects might reflect an incoherent mix of old and new values.

The CAP theorem, proposed by Brewer [17] and proved by Gilbert and Lynch [37], proves that a shared-data system that has availability and partition tolerance cannot achieve linearizability. This impossibility result can be explained quite intuitively: if a network partition occurs, availability requires that a write on one side of the partition must be able to complete, while linearizability requires that later reads on the other side of the partition must return the result of that write, which is clearly impossible.

Several forms of strong consistency, such as linearizability, serializability, and sequential consistency have been proven incompatible with low latency—defined as latency for all operations that is less than half of the maximum speed-of-light delay between replicas [10, 52]. Linearizability, for example, is easily demonstrated to be incompatible: it requires that a write to a datacenter in Asia be reflected immediately in a read to a U.S. datacenter. To achieve this property, either the write must be propagated to the U.S. site before returning, or the read must query Asia before returning. Both approaches are incompatible with the goal of low latency. In general, all consistency models with a total order over operations on multiple data locations have been shown to be incompatible with low latency. Refer to Theorem 1 of Lipton and Sandberg [52] and/or Theorem 4.1 of Attiya et al. [10] for the proof of this property.

To balance between the requirements of ALPS systems and programmability, we define an intermediate and achievable consistency model in the next section.

Chapter 3

Causal+ Consistency

A storage system's consistency guarantees can restrict the possible orderings and timing of operations throughout the system, helping to simplify the possible behaviors that a programmer must reason about and the anomalies that clients may see.

Given that the strongest forms of consistency—linearizability, serializability, and sequential consistency—are provably incompatible with our low-latency requirement [10, 52], previous partition-tolerant, low-latency systems [27, 35, 46] settled for the weakest form of consistency, eventual consistency. The commonly agreed upon definition of *eventual consistency* is that writes to one replica will eventual appear at other replicas and that if all replicas have received the same set of writes, they will have the same values for all data. This weak form of consistency does not restrict the ordering of operations on different keys in any way, forcing programmers to reason about all possible orderings and exposing many inconsistencies to users. For example, under eventual consistency, after Alice updates her profile, she might not see that update after a refresh. Or, if Alice and Bob are commenting back-and-forth on a blog post, Carol might see a random non-contiguous subset of that conversation.

Fortunately, causal+ consistency can avoid many such inconvenient orderings, including the above examples, while guaranteeing low latency.

To define *causal consistency with convergent conflict handling* (causal+ consistency), we first describe the abstract model over which it operates. There are two basic operations in our model: read and write. Data is stored and retrieved from logical *replicas*, each of which hosts the entire key space. In our work, a single logical replica corresponds to an entire local cluster of nodes.

An important concept in our model is the notion of *potential causality* [2, 47] between operations. Three rules define potential causality, denoted $\leadsto$:

1. **Thread-Of-Execution.** If a and b are two operations in a *single thread of execution*, then $a \leadsto b$ if operation a happens before operation b.[1]

2. **Reads-From.** If a is a write operation and b is a read operation that returns the value written by a, then $a \leadsto b$.

3. **Transitivity.** For operations a, b, and c, if $a \leadsto b$ and $b \leadsto c$, then $a \leadsto c$. Thus, the causal relationship between operations is the transitive closure of the first two rules.

These rules establish potential causality between operations within the same execution thread and between operations whose execution threads interacted through the data store. Our model, like many, does not allow threads to communicate directly, requiring instead that all communication occur through the data store.

An example execution of the operations in Figure 3.1(a) is shown in in Figure 3.1(b). This example demonstrates all three rules. The *thread-of-execution* rule gives $w_1 \leadsto r_4$; the *reads-from* rule gives $w_1 \leadsto r_2$; and the *transitivity* rule gives $w_1 \leadsto w_8$.

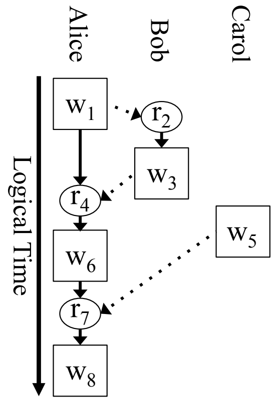
3.1 Definition

We define causal+ consistency as a combination of two properties: causal consistency and convergent conflict handling. We present intuitive definitions here and the formal definitions

[1] The original causality papers [2, 47] refer to "processes" instead of "threads of execution", we use the latter term because it is more clear in the context of today's systems.

User	Op ID	Operation
Alice	w_1	write(Alice:Town, NYC)
Bob	r_2	read(Alice:Town)
Bob	w_3	write(Bob:Town, LA)
Alice	r_4	read(Bob:Town))
Carol	w_5	write(Carol:Likes, Cats, 8/31/12)
Alice	w_6	write(Alice:Likes, Cats, 9/1/12)
Alice	r_7	read(Carol:Likes, Cats)
Alice	w_8	write(Alice:Friends, Carol, 9/2/12)

(a)



(b)

Figure 3.1: A set of example operations is shown in (a) and the graph of causality between them is shown in (b). Between operations, bold edges indicate potential causality from the thread-of-execution rule, dotted edges indicate potential causality from the reads-from rule, and a path of length greater than 1 indicates potential causality from the transitivity rule.

in Appendix A. Previous work describes systems that provide causal+ consistency [27, 32, 43, 56, 59]. Our contribution includes identifying the difference between causal and causal+, naming causal+, and explicitly defining causal+.

Causal consistency requires that values returned from read operations at a replica are consistent with the order defined by $\rightsquigarrow$ (causality) [2]. In other words, it must appear that the operation that writes a value occurs after all operations that causally precede it. For example, in Figure 3.1(b), it must appear that w_1 happened before w_3, which in turn happened before w_6. If a client saw w_6 but not w_1, causal consistency would be violated.

Causal consistency does not order concurrent operations. If $a \not\rightsquigarrow b$ and $b \not\rightsquigarrow a$, then a and b are concurrent. Normally, this allows increased efficiency in an implementation: two unrelated writes can be replicated in any order, avoiding the need for a serialization point between them. If, however, a and b are both writes to the same key, then they are in *conflict*.

Conflicts are undesirable for two reasons. First, because they are unordered by causal consistency, conflicts allow replicas to diverge forever [2]. For instance, if a writes 1 to location X and b writes 2 to location X, then causal consistency allows one replica to forever return 1 for X and another replica to forever return 2 for X. Second, conflicts may represent an exceptional condition that requires special handling. For example, in a shopping cart application, if two people logged in to the same account concurrently add items to their cart, the desired result is to end up with both items in the cart.

Convergent conflict handling requires that all conflicting writes be handled in the same manner at all replicas, using a handler function h. This handler function h must be associative and commutative, so that replicas can handle conflicting writes in the order they receive them and that the results of these handlings will converge (e.g., one replica's $h(a,h(b,c))$ and another's $h(c,h(b,a))$ agree).

One common way to handle conflicting writes in a convergent fashion is the last-writer-wins rule (also called Thomas's write rule [72]), which declares one of the conflicting writes as having occurred later and has it overwrite the "earlier" write. Another common way to handle conflicting writes is to mark them as conflicting and require their resolution by some other means, e.g., through direct user intervention as in Coda [43], or through a programmed procedure as in Bayou [59] and Dynamo [27].

All potential forms of convergent conflict handling avoid the first issue—conflicting updates may continually diverge—by ensuring that replicas reach the same result after exchanging operations. On the other hand, the second issue with conflicts—applications may want special handling of conflicts—is only avoided by the use of more explicit *conflict resolution* procedures. These explicit procedures provide greater flexibility for applications, but require additional programmer complexity and/or performance overhead. Although our systems can be configured to detect conflicting updates explicitly and apply some application-defined resolution, the default versions of our systems use the last-writer-wins rule.

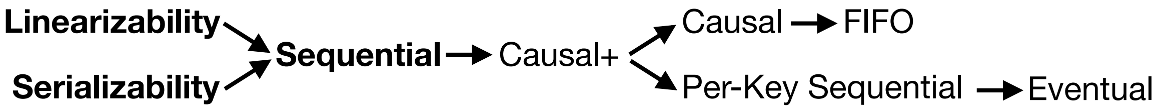


Figure 3.2: A spectrum of consistency models, with stronger models on the left. Bolded models are provably incompatible with ALPS systems.

3.2 Causal+ vs. Other Consistency Models

The distributed systems literature defines several popular consistency models. They include: linearizability (or strong consistency) [41], which maintains a global, real-time ordering; serializability [13], which ensures a global ordering of transactions; sequential consistency [48], which ensures a global ordering of operations; causal consistency [2], which ensures partial orderings between dependent operations; FIFO (PRAM) consistency [52], which only preserves the partial ordering of an execution thread, not between threads; per-key sequential consistency [25], which ensures that, for each individual key, all operations have a global order; and eventual consistency, a "catch-all" term used today to suggest eventual convergence to some type of agreement.

The causal+ consistency we introduce falls between sequential and causal consistency, as shown in Figure 3.2. It is weaker than sequential consistency, but sequential consistency is provably not achievable in an ALPS system. It is stronger than causal consistency and per-key sequential consistency, and is achievable for ALPS systems.[2]

To illustrate the utility of the causal component of the causal+ model, consider an example where a user uploads a photo to an Internet service and then adds that photo to an album. Under causal and thus causal+ consistency, the photo will always arrive before the add-to-album operation. Under per-key sequential and eventual consistency, it is possible for the add-to-album operation to appear before the photo. This forces programmers to reason about that scenario: should they drop the add-to-album operation, queue up a callback for when the photo arrives, add a broken reference to the album, or do something else? The

[2] Mahajan et al. [55] have concurrently defined a similar strengthening of causal consistency; see Section 8 for details.

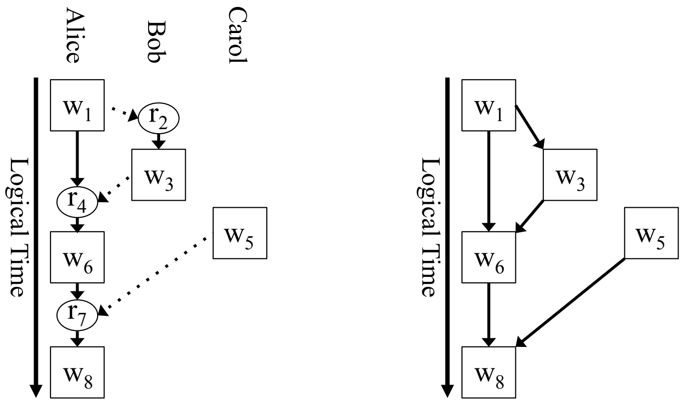
generalization of this scenario where operation B appears before a proceeding operation A maps to many situations where user expectations are not obeyed and/or programmers must reason about out-of-order operations. Some of these other situations include removing a boss from a friend's list (A) and then posting about seeking a new job (B); reading a friend's status update (A) and then replying to it (B); adding an item to a shopping cart (A) and then checking to see if it is there (B); a status update (A) followed by trying to hide it (B); or an account creation (A) followed by trying to log in (B).

To illustrate the utility of the convergent conflict handling component of the causal+ model, consider an example where Alice and Bob both update the starting time for an event. The time was originally set for 9pm, Alice changed it to 8pm, and Bob concurrently changed it to 10pm. Regular causal consistency would allow two different replicas to forever return different times, even after receiving both write operations. Causal+ consistency requires that replicas handle this conflict in a convergent manner. If a last-writer-wins policy is used, then either Bob's 10pm or Alice's 8pm would win. If a more explicit conflict resolution policy is used, the key could be marked as in conflict and future reads on it could return both 8pm and 10pm with instructions to resolve the conflict.

If the data store was sequentially consistent or linearizable, it would still be possible for there to be two simultaneous updates to a key. In these stronger models, however, it is possible to implement mutual exclusion algorithms—such as the one suggested by Lamport in the original sequential consistency paper [48]—that can be used to avoid creating a conflict altogether.

3.3 Causal+ in Practice

We use two abstractions in our systems, *versions* and *dependencies*, to help us reason about causal+ consistency. We refer to the different values a data location has as the *versions* of a location, which we denote $\text{loc}_{\text{version}}$. In our systems, versions are assigned in a manner that



(a) (b)

Figure 3.3: A graph of causality is shown in (a) and the corresponding dependency graph is shown in (b).

ensures that if $x_i \rightsquigarrow y_j$ then $i < j$. Once a replica returns version i of a location, x_i, causal+ consistency ensures it will then only return that version or a causally later version (note that the handling of a conflict is causally later than the conflicting writes it resolves).[3] Thus, each replica in our systems always returns non-decreasing versions of a key. We refer to this as causal+ consistency's *progressing property*.

Causal consistency dictates that all operations that causally precede a given operations must appear to take effect before it. In other words, if $x_i \rightsquigarrow y_j$, then x_i must be written before y_j. We call these preceding values *dependencies*. More formally, we say y_j ***depends on*** x_i if and only if $\texttt{write}(x_i) \rightsquigarrow \texttt{write}(y_j)$. These dependencies are the reverse of the causal ordering of writes and are, by definition, the same as the happens-before relationship [47]. Figure 3.3(a) shows a graph of causality between operations (this is the same graph as Figure 3.1(b)) and Figure 3.3(b) shows the corresponding dependencies between the write operations.

[3] To see this, consider by contradiction the following scenario: assume a replica first returns x_i and then x_k, where $i \neq k$ and $x_i \not\rightsquigarrow x_k$. Causal consistency ensures that if x_k is returned after x_i, then $x_k \not\rightsquigarrow x_i$, and so x_i and x_k conflict. But, if x_i and x_k conflict, then convergent conflict handling ensures that as soon as both are present at a replica, their handling $h(x_i,x_k)$, which is causally after both, will be returned instead of either x_i or x_k, which contradicts our assumption.

Our systems provide causal+ consistency during replication by writing a version only after all of its dependencies are *satisfied*, meaning those writes have already been applied in the cluster.

3.4 Scalable Causality

To our knowledge, our work is the first to name and formally define causal+ consistency. Interestingly, several previous systems [12, 59] believed to achieve causal consistency in fact achieved the stronger guarantees of causal+ consistency.

These systems were not designed to and do not provide *scalable* causal (or causal+) consistency, however, as they all use a form of log serialization and exchange. All operations at a logical replica are written to a single log in serialized order, commonly marked with a version vector [58]. Different replicas then exchange these logs, using version vectors to establish potential causality and detect concurrency between operations at different replicas.

Log-exchange-based serialization impairs replica scalability, as it relies on a single serialization point in each replica to establish ordering. Thus, either causal dependencies between keys are limited to the set of keys that can be stored on one node [12, 25, 46, 59], or a single node (or replicated state machine) must provide a commit ordering and log for all operations across a cluster.

As we show later, our systems achieve scalability by taking a different approach. Nodes in each datacenter are responsible for different partitions of the keyspace, but the system can track and enforce dependencies between keys stored on different nodes. Our systems explicitly encode dependencies in metadata associated with each key's version. When keys are replicated remotely, the receiving datacenter performs dependency checks before committing the incoming version. They perform these checks by communicating directly with the nodes responsible for storing the other keys. In this way, they provide the first, to our knowledge, mechanisms for achieving scalable causal+ replication.

Chapter 4

System Design of COPS

COPS [53] is a distributed storage system that realizes causal+ consistency and possesses the desired ALPS properties. There are two distinct versions of the system. The first, which we refer to simply as COPS, provides a data store that is causal+ consistent. The second, called COPS-RT, provides a superset of this functionality by also introducing support for *read-only transactions*. With read-only transactions, clients request a set of keys and the data store replies with a consistent and up-to-date snapshot of corresponding values. Because of the additional metadata needed to enforce the consistency properties of read-only transactions, a given deployment must run exclusively as COPS or COPS-RT.

Assumptions. The design of COPS assumes an underlying sharded, reliable, and linearizable data store inside of each datacenter. Specifically, we assume:

1. The keyspace is sharded across logical servers.

2. Linearizability is provided inside a datacenter.

3. Keys are stored on logical servers, implemented with replicated state machines. We assume that a failure does not make a logical server unavailable, unless it makes the entire datacenter unavailable.

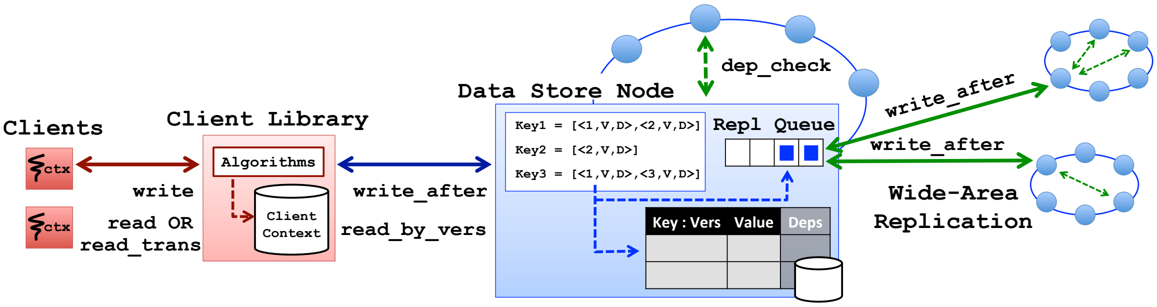


Figure 4.1: The COPS architecture. A client library exposes a read/write interface to its clients and ensures operations are properly labeled with causal dependencies. COPS exposes the `read` operation to clients, while COPS-RT exposes `read_trans`. A key-value store replicates data between clusters, ensures writes are committed in their local cluster only after their dependencies have been satisfied, and in COPS-RT, stores multiple versions of each key along with dependency metadata.

Each assumption represents an orthogonal direction of research to our work. By assuming these properties instead of specifying their exact design, we focus our explanation on the novel facets of our work.

Keyspace partitioning may be accomplished with consistent hashing [42] or directory-based approaches [7, 36]. Linearizability within a datacenter is achieved by partitioning the keyspace and then providing linearizability for each partition [41]. Linearizability is acceptable locally because we expect very low latency and no partitions within a cluster—especially with the trend towards redundant paths in modern datacenter networks [3, 39]—unlike in the wide-area. Reliable, linearizable servers can be implemented with Paxos [49] or primary-backup [4] approaches, e.g., chain replication [74]. Many existing systems [6, 16, 20, 70], in fact, provide all assumed properties when used inside a single datacenter.

4.1 Overview of COPS

COPS is a key-value storage system designed to run across a small number of datacenters, as illustrated in Figure 4.1. Each datacenter has a local COPS *cluster* with a complete replica of the stored data.[1] A *client* of COPS is an application that uses the COPS *client library*

[1] As mentioned earlier, investigating partial replication is an exciting area of future research.

to call directly into the COPS key-value store. Clients communicate only with their local COPS cluster running in the same datacenter.

System Components. COPS is composed of two main software components:

- *Key-value store.* The basic building block in COPS is a standard key-value store that provides linearizable operations on keys. COPS extends the standard key-value store in two ways, and COPS-RT adds a third extension.

  1. Each key-value pair has associated metadata. In COPS, this metadata is a timestamp. In COPS-RT, it is both a timestamp and a list of dependencies (other keys and their respective timestamps).
  2. The key-value store exports three additional operations as part of its key-value interface: `read_by_time`, `write_after`, and `dep_check`, each described below. These operations enable the COPS client library and an asynchronous replication process that supports causal+ consistency and read transactions.
  3. For COPS-RT, the system keeps around old versions of key-value pairs, not just the most recent `write`, to ensure that it can provide read transactions. Maintaining old versions is discussed further in Section 4.3.

- *Client library.* The client library exports two main operations to applications: reads via `read` (in COPS) or `read_trans` (in COPS-RT), and writes via `write`. The client library also maintains state about a client's current dependencies through an *actor_id* parameter in the client library API.

Goals. The COPS design strives to provide causal+ consistency with resource and performance overhead similar to existing eventually-consistent systems. COPS and COPS-RT must therefore:

- *Minimize overhead of consistency-preserving replication.* COPS must ensure that values are replicated between clusters in a causal+ consistent manner. A naive implementation, however, would require checks on all of a value's dependencies. We present a mechanism that requires only a small number of such checks by leveraging the graph structure inherent to causal dependencies.

- *(COPS-RT) Minimize space requirements.* COPS-RT stores (potentially) multiple versions of each key, along with their associated dependency metadata. COPS-RT uses aggressive garbage collection to prune old state (see Section 6.1).

- *(COPS-RT) Ensure fast `read_trans` operations.* The read transactions in COPS-RT ensure that the set of returned values are causal+ consistent (all dependencies are satisfied). A naive algorithm could block and/or take an unbounded number of read rounds to complete. Both situations are incompatible with the ALPS goals of availability and low latency; we present an algorithm for `read_trans` that completes in at most two rounds of local `read_by_time` operations.

4.2 The COPS Key-Value Store

Unlike traditional ⟨*key*, *value*⟩–tuple stores, COPS must track the timestamps of written values, and COPS-RT must additionally track their dependencies. In COPS, the system stores the most recent timestamp and value for each key. In COPS-RT, the system maps each key to a list of versions, each consisting of ⟨*timestamp*, *value*, *deps*⟩. The *deps* field is a list of the version's zero or more dependencies; each dependency is a ⟨*key*, *timestamp*⟩ pair.

Each COPS cluster maintains its own copy of the key-value store. Operations return to the client library as soon as they execute in the local cluster; operations between clusters occur asynchronously.

Every key stored in COPS is stored in a logical server (server). We term the set of servers for a key across all clusters as the *equivalent servers* for that key. In practice, COPS's consistent hashing assigns each server a few different key ranges. Key ranges may have different sizes and node mappings in different datacenters, but the total number of equivalent nodes with which a given node needs to communicate is proportional to the number of datacenters (i.e., communication is not all-to-all between nodes in different datacenters).

After a write completes locally, the server places it in a replication queue, from which it is sent asynchronously to remote equivalent servers. Those nodes, in turn, wait until the value's dependencies are satisfied in their local cluster before locally committing the value. This dependency checking mechanism ensures writes happen in a causally consistent order and reads never block.

4.3 Client Library and Interface

The COPS client library provides a simple and intuitive programming interface with two operations:

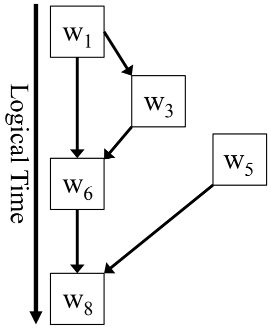
1. bool ← `write` (*key*, *value*, *actor_id*)

2. *value* ← `read` (*key*, *actor_id*) [In COPS]

or

2. ⟨*values*⟩ ← `read_trans` (⟨*keys*⟩, *actor_id*) [In COPS-RT]

The client API differs from a traditional key-value interface in two ways. First, COPS-RT provides `read_trans`, which returns a consistent view of multiple key-value pairs in a single call. Second, all functions take an *actor_id* argument, which the library uses internally to track causal dependencies across each client's operations [71]. The *actor_id* defines the causal+ "thread of execution." A single process may contain many separate threads of execution (e.g., a web server concurrently serving 1000s of independent connections). By separating different threads of execution, COPS avoids false dependencies that would result from intermixing them.



(a)

Op	Dependencies
w_1	-
w_3	$\underline{\boldsymbol{w_1}}$
w_5	-
w_6	$\underline{\boldsymbol{w_3}}$ $\underline{w_1}$
w_8	$\underline{\boldsymbol{w_6}}$ $\underline{\boldsymbol{w_5}}$ w_3 w_1

(b)

Figure 4.2: A dependency graph is shown in (a) and the table in (b) lists nearest (bold), one-hop (underlined), and all dependencies.

We first describe the state kept by the client library in COPS-RT to enforce consistency in read transactions and then show how COPS can store significantly less dependency state.

COPS-RT Client Library. The client library in COPS-RT stores the client's context in a table of $\langle$*key*, *timestamp*, *deps*$\rangle$ entries. Clients reference their context using an *actor_id* in the API.[2] When a client reads a key from the data store, the library adds this key and its causal dependencies to the context. When a client writes a value, the library sets the write's dependencies to the most recent timestamp of each key in the current context. A successful write into the data store returns the timestamp number *ts* assigned to the written value. The client library then adds this new entry, $\langle$*key*, *ts*, *deps*$\rangle$, to the context.

The context therefore includes dependencies on all previous writes in the current session (thread-of-execution rule), all writes that wrote data that was read during the session (reads-from rule), and all of those dependencies' dependencies (transitivity rule), as illustrated in Figure 4.2(a). This raises two concerns about the potential size of this causality graph: (i) state requirements for storing these dependencies, both in the client library and in the

[2] Maintaining state in the library and passing in an ID was a design choice; one could also encode the entire context table as an opaque blob and pass it between client and library so that the library is stateless.

data store, and (ii) the number of potential checks that must occur when replicating writes between clusters, in order to ensure causal consistency.

Fortunately, we can leverage the structure of the dependency graph to mitigate both of these concerns by only tracking and enforcing a small subset of all dependencies that we call the *nearest dependencies*.[3] These dependencies, which have a longest path of one hop to the current operation, transitively capture all of the ordering constraints on this operation. In particular, because all non-nearest dependencies are depended upon by at least one of the nearest, if the current operation occurs after the nearest dependencies, then it will occur after all non-nearest as well by transitivity.

COPS (and Eiger) actually track *one-hop dependencies*, a slightly larger superset of nearest dependencies, which have a shortest path of one hop to the current operation. We elect to track one-hop dependencies because we can do so without storing any dependency information at the servers. Using one-hop dependencies slightly increases both the amount of memory needed at the client nodes and the data sent to servers on writes. In contrast, our alternative design for tracking the slightly smaller set of nearest dependencies put the dependency storage burden on the servers, a trade-off we did not believe generally worthwhile. Figure 4.2(b) illustrates the types of dependencies, e.g., w_6's dependency on w_1 is one-hop but not nearest.

Checking nearest (and thus one-hop) dependencies are sufficient for the key-value store to provide causal+ consistency; the full dependency list is only needed to provide `read_trans` operations in COPS-RT.

COPS Client Library. The client library in COPS requires significantly less state and complexity because it only needs to learn the one-hop, rather than all, dependencies. Accordingly, it does not store or even retrieve the dependencies of any value it reads: the retrieved value is nearer than any of its dependencies, rendering them unnecessary.

[3] In graph theoretic terms the nearest dependencies are called the predecessor set of the *transitive reduction* of the dependency graph. We use the term nearest dependencies in this dissertation for consistency with our earlier work.

Thus, the COPS client library stores only $\langle$*key*, *timestamp*$\rangle$ entries. For a `read` operation, the retrieved $\langle$*key*, *timestamp*$\rangle$ is added to the context. For a write operation, the library uses the current context as the nearest dependencies, clears the context, and then repopulates it with only this write. This write depends on all previous key-timestamp pairs and thus is nearer than them.

4.4 Writing Values in COPS and COPS-RT

Building on our description of the client library and key-value store, we now walk through the steps involved in writing a value to COPS. All writes in COPS first go to the client's local cluster and then propagate asynchronously to remote clusters. The key-value store exports a single API call to provide both operations:

$\langle$bool,*timestamp*$\rangle \leftarrow$ `write_after` (*key*, *value*, *one_hop*, *timestamp*=$\emptyset$) [In COPS]

or

$\langle$bool,*timestamp*$\rangle \leftarrow$ `write_after` (*key*, *value*, *deps*, *nearest*, *timestamp*=$\emptyset$) [In COPS-RT]

Writes to the local cluster. When a client calls `write` (*key*,*value*,*actor_id*), the library computes what dependencies to attach and then calls `write_after` without the *timestamp* argument (i.e., it sets *timestamp*=$\emptyset$). The attached dependencies vary significantly between COPS and COPS-RT. COPS-RT requires all dependencies for read-only transactions, but only need to check nearest dependencies. COPS-RT can calculate nearest dependencies because it has the full causality graph. For example, in Figure 4.2(a) COPS-RT can see the edge from w_1 to w_3 and knows w_1 is not a nearest dependency for w_6. COPS, however, can only calculate the one-hop dependencies because it does not store any dependencies on servers and thus would never see the edge from w_1 to w_3.

The key's server in the local cluster assigns the key a timestamp and returns it to the client library. We restrict each client to a single outstanding write; this is necessary because later writes must know the timestamps of earlier writes so they may depend on them.

The `write_after` operation ensures that *value* is committed to each cluster only after all of the entries in its dependency list have been written. In the client's local cluster, this property holds automatically, as the local store provides linearizability. (If y depends on x, then `write`(x) must have been committed before `write`(y) was issued.) This is not true in remote clusters, however, which we discuss below.

Clients and servers in COPS maintain logical clocks [47], all messages include a logical timestamp that updates these clocks, and written values are assigned a timestamp based on logical time.[4] The clocks and timestamps provide a progressing logical time throughout the entire system. The timestamps for written values have additional low-order bits appended that are set to the stamping server's unique identifier, so each is globally distinct.

These logical timestamps allow COPS to derive a single global order over all writes for each individual key. This order implicitly implements the last-writer-wins convergent conflict handling policy. COPS is also capable of explicitly detecting and resolving conflicts, which we discuss in Section 6.3. Note that because Lamport timestamps provide a partial ordering of all distributed events in a way that respects potential causality, this global ordering is compatible with COPS's causal consistency.

Write replication between clusters. After a write commits locally, the server asynchronously replicates that write to its equivalent nodes in different clusters using a stream of `write_after` operations; here, however, the server includes the key's timestamp number in the `write_after` call. This approach scales well and avoids the need for a single serialization point, but requires the remote nodes receiving updates to commit an update only after its dependencies have been committed to the same cluster.

[4] With logical (Lamport) clocks, each node in the system keeps a logical clock that is updated every time an event occurs (e.g., writing a value or receiving a message). When sending a message, a node includes a timestamp t set to its logical clock c; when receiving a message, a node sets $c \leftarrow max(c, t+1)$.

To ensure this property, a node that receives a `write_after` request from another cluster must determine if the value's nearest (COPS-RT) or one-hop (COPS) dependencies have already been satisfied locally. It does so by issuing a check to the local nodes responsible for the those dependencies:

bool ← `dep_check` (*key*, *timestamp*)

When a node receives a `dep_check`, it examines its local state to determine if the dependency value has already been written. If so, it immediately responds to the operation. If not, it blocks until the needed timestamp has been written.

If all `dep_check` operations on the nearest dependencies succeed, the node handling the `write_after` request commits the written value, making it available to other reads and writes in its local cluster. (If any `dep_check` operation times out, the node handling the `write_after` reissues it, potentially to a new node if a failure occurred.) The way that nearest/one-hop dependencies are computed ensures that all dependencies have been satisfied before the value is committed, which in turn ensures causal consistency.

4.5 Reading Values in COPS

Like writes, reads are satisfied in the local cluster. Clients call the `read` library function with the appropriate *actor_id*; the library in turn issues a read to the node responsible for the key in the local cluster:

⟨*value*, *timestamp*, *deps*⟩ ← `read_by_time` (*key*, *timestamp*=`LATEST`)

This read can request either the latest timestamp of the key or a specific older one. Requesting the latest timestamp is equivalent to a regular single-key `read`; requesting a specific timestamp is necessary to enable read transactions. Accordingly, `read_by_time` operations in COPS always request the latest timestamp. Upon receiving a response, the client library adds the $\langle$*key*,*timestamp*[,*deps*]$\rangle$ tuple to the client context, and returns *value* to the calling code. The *deps* are stored only in COPS-RT, not in COPS.

4.6 Read-only Transactions in COPS-RT

The COPS-RT client library provides a `read_trans` interface because reading a set of dependent keys using a single-key `read` interface cannot ensure causal+ consistency, even though the data store itself is causal+ consistent. Intuitively, this is because asynchronous requests to distributed servers will not arrive at the servers, or be responded to, simultaneously. Concretely, we demonstrate this problem by extending the photo album example to include access control, whereby Alice first changes her album ACL to “friends only”, and then writes a new description of her travels and adds more photos to the album.

A natural (but incorrect) implementation of code to read Alice’s album might (1) fetch the ACL, (2) check permissions, and (3) fetch the album description and photos. This approach contains a straightforward “time-to-check-to-time-to-use” race condition: when Eve accesses the album, her `read`(ACL) might return the old ACL, which permitted anyone (including Eve) to read it, but her `read`(album contents) might return the “friends only” version.

One straw-man solution is to require that clients issue single-key `read` operations in the reverse order of their causal dependencies: The above problem would not have occurred if the client executed `read`(album) before `read`(ACL). This solution, however, is also incorrect. Imagine that after updating her album, Alice decided that some photographs were too personal, so she (3) deleted those photos and rewrote the description, and then (4) marked the ACL open again. This straw-man has a different time-of-check-to-time-of-use error, where `read`(album) retrieves the private album, and the subsequent `read`(ACL) retrieves the “public” ACL. In short, there is no correct canonical ordering of the ACL and the album entries.

```
# @param  keys    list of keys
# @param  ctx_id  context id
# @return values  list of values

function get_trans(keys, ctx_id):
  # Get keys in parallel (first round)
  for k in keys
    results[k] = get_by_version(k, LATEST)

  # Calculate causally correct versions (ccv)
  for k in keys
    ccv[k] = max(ccv[k], results[k].vers)
    for dep in results[k].deps
      if dep.key in keys
        ccv[dep.key] = max(ccv[dep.key], dep.vers)

  # Get needed ccvs in parallel (second round)
  for k in keys
    if ccv[k] > results[k].vers
      results[k] = get_by_version(k, ccv[k])

  # Update the metadata stored in the context
  update_context(results, ctx_id)

  # Return only the values to the client
  return extract_values(results)
```

Figure 4.3: Pseudocode for the read_trans algorithm.

Instead, a better programming interface would allow the client to obtain a causal+ consistent view of multiple keys. The standard way to achieve such a guarantee is to read and write all related keys in a transaction; this, however, requires a single serialization point for all grouped keys, which COPS avoids for greater scalability and simplicity. Instead, COPS allows keys to be written independently (with explicit dependencies in metadata), and provides a read_trans operation for retrieving a consistent view of multiple keys.

Read-only transactions. To retrieve multiple values in a causal+ consistent manner, a client calls read_trans with the desired set of keys, e.g., read_trans(⟨ACL, album⟩).

Depending on when and where it was issued, this read transaction can return different combinations of ACL and album, but never $\langle \text{ACL}_{\text{public}}, \text{Album}_{\text{personal}} \rangle$.

The COPS client library implements the read transactions algorithm in two rounds, shown in Figure 4.3. In the first round, the library issues n concurrent `read_by_time` operations to the local cluster, one for each key the client listed in `read_trans`. Because COPS-RT commits writes locally, the local data store guarantees that each of these explicitly listed keys' dependencies are already satisfied—that is, they have been written locally and reads on them will immediately return. These explicitly listed, independently retrieved values, however, may not be consistent with one another, as shown above. Each `read_by_time` operation returns a $\langle$*value*, *timestamp*, *deps*$\rangle$ tuple, where *deps* is a list of keys and timestamps. The client library then examines every dependency entry $\langle$*key*, *timestamp*$\rangle$. The causal dependencies for that result are satisfied if either the client did not request the dependent key, or if it did, the timestamp it retrieved was $\geq$ the timestamp in the dependency list.

For all keys that are not satisfied, the library issues a second round of concurrent `read_by_time` operations. The timestamp requested will be the newest timestamp seen in any dependency list from the first round. These timestamps satisfy all causal dependencies from the first round because they are $\geq$ the needed timestamps. In addition, because dependencies are transitive and these second-round timestamps are all depended on by timestamps retrieved in the first round, they do not introduce any new dependencies that need to be satisfied. This algorithm allows `read_trans` to return a consistent view of the data store as of the time of the latest timestamp retrieved in first round.

The second round happens only when the client must read newer timestamps than those retrieved in the first round. This case occurs only if keys involved in the read transaction are updated during the first round. Thus, we expect the second round to be rare. In our example, if Eve issues a `read_trans` concurrent with Alice's writes, the algorithm's first round of reads might retrieve the public ACL and the private album. The private album, however, depends on the “friends only” ACL, so the second round would fetch this newer timestamp of the ACL, allowing `read_trans` to return a causal+ consistent set of values to the client.

The causal+ consistency of the data store provides two important properties for the read transaction algorithm’s second round. First, the `read_by_time` requests will succeed immediately, as the requested timestamp must already exist in the local cluster. Second, the new `read_by_time` requests will not introduce any new dependencies, as those dependencies were already known in the first round due to transitivity. This second property demonstrates why the read transaction algorithm specifies an explicit timestamp in its second round, rather than just getting the latest: otherwise, in the face of concurrent writes, a newer timestamp could introduce still newer dependencies, which could continue indefinitely.

Chapter 5

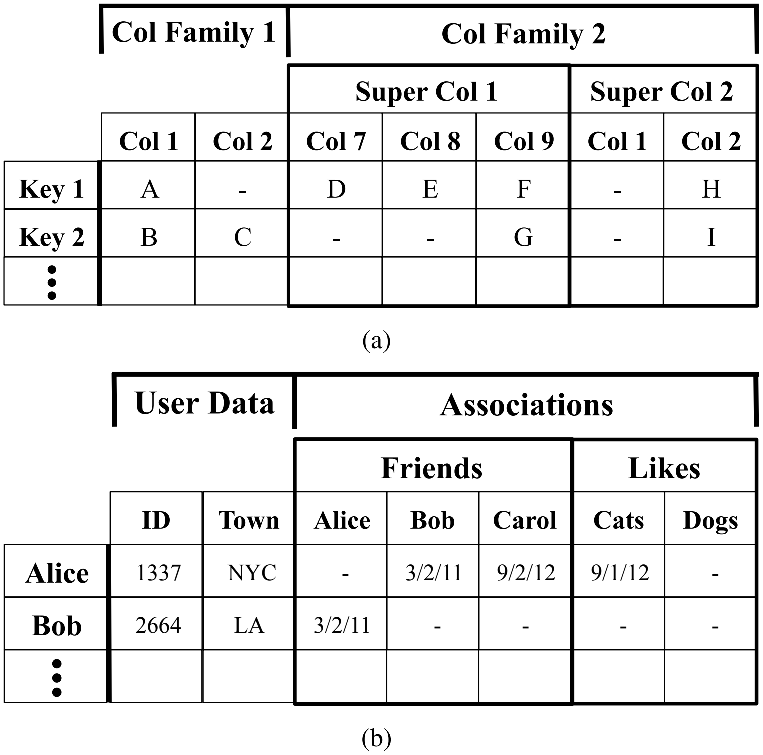
System Design of Eiger

Eiger supersedes both COPS and COPS-RT. Eiger provides read-only transactions like COPS-RT, but has minimal metadata like COPS. It also operates on a richer data model than COPS and supports write-only transactions. Eiger make the same assumptions as COPS: The keyspace is partitioned across logical servers; Linearizability is provided inside a datacenter; and Keys are stored on logical servers, implemented with replicated state machines.

We describe the data model Eiger operates on, detail the differences needed in its architecture to support causal+ consistency for that data model, describe the new algorithm for read-only transactions, and then present write-only transactions.

5.1 Column-Family Data Model

Eiger uses the column-family data model, which provides a rich structure that allows programmers to naturally express complex data and then efficiently query it. This data model was pioneered by Google's BigTable [23]. It is now available in the open-source Cassandra system [46], which is used by many large web services including EBay, Netflix, and Reddit.



	Col Family 1		Col Family 2				
			Super Col 1			Super Col 2	
	Col 1	Col 2	Col 7	Col 8	Col 9	Col 1	Col 2
Key 1	A	-	D	E	F	-	H
Key 2	B	C	-	-	G	-	I
⋮							

(a)

	User Data		Associations				
			Friends			Likes	
	ID	Town	Alice	Bob	Carol	Cats	Dogs
Alice	1337	NYC	-	3/2/11	9/2/12	9/1/12	-
Bob	2664	LA	3/2/11	-	-	-	-
⋮							

(b)

Figure 5.1: The column family data model in (a), as well as an example use of the model for a social network setting in (b).

Our implementation of Eiger is built upon Cassandra and so our description adheres to its specific data model where it and BigTable differ.[1] Our description of the data model and API are simplified, when possible, for clarity.

Basic Data Model. The column-family data model is a "map of maps of maps" of named columns. The first-level map associates a key with a set of named column families. The second level of maps associates the column family with a set composed exclusively of either columns or super columns. If present, the third and final level of maps associates each super column with a set of columns. Figure 5.1(a) shows this structure abstractly and Figure 5.1(b) gives an example where it could be used for a social network.

[1] For example, BigTable columns can include multiple timestamped versions of each value, while Cassandra columns include only a single value. On the other hand, Cassandra's data model includes super columns, which BigTable does not.

`bool`	⟵	`batch_mutate`	`( {key→mutation}, actor_id )`
`bool`	⟵	`atomic_mutate`	`( {key→mutation}, actor_id )`
`{key→columns}`	⟵	`multiget_slice`	`( {key, column_parent, slice_predicate}, actor_id )`

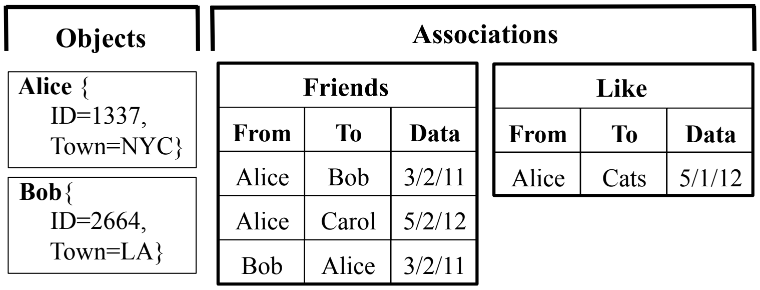
Table 5.1: Core API functions in Eiger's column family data model. Eiger introduces `atomic_mutate` and converts `multiget_slice` into a read-only transaction.

Within a column family, each *location* is represented as a compound key and a single value, i.e., "Alice:Assocs:Friends:Bob" with value "3/2/11". These pairs are stored in a simple ordered key-value store. All data for a single row must reside on the same server.

Clients use the API shown in Table 5.1. Clients can insert, update, or delete columns for multiple keys with a `batch_mutate` or an `atomic_mutate` operation; each mutation is either an `insert` or a `delete`. If a column exists, an `insert` updates the value. Mutations in a `batch_mutate` appear independently, while mutations in an `atomic_mutate` appear as a single atomic group.

Similarly, clients can read many columns for multiple keys with the `multiget_slice` operation. The client provides a list of tuples, each involving a key, a column family name and optionally a super column name, and a slice predicate. The slice predicate can be a `(start,stop,count)` three-tuple, which matches the first `count` columns with names between `start` and `stop`. Names may be any comparable type, e.g., strings or integers. Alternatively, the predicate can also be a list of column names. In either case, a *slice* is a subset of the stored columns for a given key.

Given the example data model in Figure 5.1 for a social network, the following function calls show three typical API calls: updating Alice's hometown when she moves, ending Alice and Bob's friendship, and retrieving up to 10 of Alice's friends with names starting with B to Z.



Objects

Alice {
ID=1337,
Town=NYC}

Bob{
ID=2664,
Town=LA}

Associations

Friends

From	To	Data
Alice	Bob	3/2/11
Alice	Carol	5/2/12
Bob	Alice	3/2/11

Like

From	To	Data
Alice	Cats	5/1/12

Figure 5.2: An example of Facebook's data model in TAO. This data translates into the column family data model shown in Figure 5.1.

```
batch_mutate ( Alice→insert(UserData:Town=Rome) )

atomic_mutate ( Alice→delete(Assocs:Friends:Bob),
                Bob→delete(Assocs:Friends:Alice) )

multiget_slice ( {Alice, Assocs:Friends, (B, Z, 10)} )
```

Counter Columns. Standard columns are updated by `insert` operations that overwrite the old value. Counter columns, in contrast, can be commutatively updated using an `add` operation. They are useful for maintaining numerical statistics, e.g., a "liked_by_count" for Cats (not shown in figure), without the need to carefully read-modify-write the object.

Why target the column family model? Its wide popularity in Google, Facebook, and elsewhere suggests that the column family model meets the needs of many rich services such as online social networks. For example, the social graph API that Facebook's front-end web programmers use can be converted to, and efficiently implemented by, the column family data model.

We reproduce Facebook's graph data model in Figure 5.2. It represents each object (vertex) using a map, and records typed associations (edges) from one object to another [33]. The map for each object can be realized with a column family that contains a column for each map entry. Similarly, the associations can be realized with super column for each type, and a column under that super column for each association from that key.[2]

5.2 Causal+ Consistency for Column-Families

Eiger operates similarly to COPS: at a high-level it tracks one-hop dependencies and ensures they are satisfied before applying replicated writes. Eiger does not use the dependency graph for read-only transactions and thus avoids the complications and inefficiencies of the COPS-RT system. Here we detail the differences between Eiger and COPS necessary to provide causal+ consistency for the column-family data model. The differences include having dependencies on operations, the types of write operations, the results of read operations, splitting multi-server operations, and counter columns.

Dependencies on Operations. Tracking dependencies on operations significantly improves Eiger's efficiency. In the column-family data model, it is not uncommon to simultaneously read or write many columns for a single key. With dependencies on values (the original and published design for COPS), a separate dependency must be used for each column's value and thus |column| dependency checks would be required; Eiger could check as few as one. In the worst case, when all columns were written by different operations, the number of required dependency checks degrades to one per value.

Dependencies in Eiger consist of a locator and a timestamp. The *locator* is used to ensure that any other operation that depends on this operation knows which node to check with to determine if the operation has been committed. For mutations of individual keys, the locator is simply the key itself. Within a write transaction the locator can be any key in the set; all that matters is that each "sub-operation" within an atomic write be labeled with the

[2] This graph model is not unlike the link graph example described in Google's original BigTable paper, which also maps well to the column family data model, by presumed design.

same locator. The timestamp, as in COPS, is globally unique and is used by servers that receive a `dep_check` to know what operation is being indicated.

New Types of Writes. There are three types of write operations in Eiger—`insert`, `add`, and `delete`—and each operates by replacing the current (potentially non-existent) column in a location. `insert` overwrites the current value with a new column (this is similar to `write` in COPS), e.g., update Alice's home town from NYC to MIA. `add` merges the current counter column with the update, e.g., increment a liked-by count from 8 to 9. `delete` overwrites the current column with a tombstone, e.g., Carol is no longer friends with Alice. When each new column is written, it is *timestamped* with the current logical time at the server applying the write. Cassandra atomically applies updates to a single row using snap trees [18], so all updates to a single key in a `batch_mutate` have the same timestamp. Updates to different rows on the same server in a `batch_mutate` will have different timestamps because they are applied at different logical times.

New Results for Reads. Read operations return the current column for each requested location. Normal columns return binary data. Deleted columns return an empty column with a deleted bit set. The client library strips deleted columns out of the returned results, but records dependencies on them as required for correctness. Counter columns return a 64-bit integer.

Splitting Operations. `batch_mutate` operations update keys that may be spread across many different servers. Thus, sometimes the client library must split `batch_mutate` operations that span multiple server into a set of `batch_mutate` operations, one per server. The client library issues these in parallel and waits for acknowledgment from all of them before returning to the client. Similarly, because the keyspace partitioning may vary from datacenter to datacenter, a replicating server must sometimes also split `batch_mutate` operations.

Counter Columns. The commutative nature of counter columns complicates tracking dependencies. In normal columns with overwrite semantics, each value was written by exactly one operation. In counter columns, each value was affected by many operations. Consider a counter with value 7 from +1, +2, and +4 operations. Each operation contributed to the final value, so a read of the counter incurs dependencies on all three. Eiger stores these dependencies with the counter and returns them to the client, so they can be attached to its next write.

Naively, every update of a counter column would increment the number of dependencies contained by that column *ad infinitum*. To bound the number of contained dependencies, Eiger structures the `add` operations occurring within a datacenter. All `add` operations originating within a datacenter are already ordered because individual datacenters are linearizable. Eiger explicitly tracks this ordering in a new `add` by adding an *extra dependency* on the previously accepted `add` operation from the datacenter. This creates a single dependency chain that transitively covers all previous updates from the datacenter. As a result, each counter column contains at most one dependency per datacenter.

Eiger further reduces the number of dependencies contained in counter columns to the nearest dependencies within that counter column. When a server applies an `add`, it examines the operation's attached dependencies. It first identifies all dependencies that are on updates from other datacenters to this counter column. Then, if any of those dependencies match the stored dependency for another datacenter, Eiger drops the stored dependency. The new operation is causally after any local matches, and thus a dependency on it transitively covers those matches as well. For example, if Alice reads a counter with the value 7 and then increments it to 8, her increment is causally after all the operations that commuted to create the value she read. Thus, any reads of the resulting 8 would only bring a dependency on Alice's update.

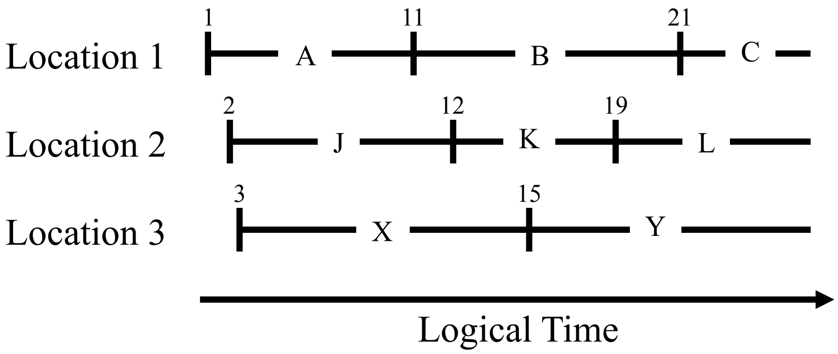


Figure 5.3: Validity periods for values written to different locations. Crossbars (and the specified numeric times) correspond to the earliest and latest valid time for values, which are represented by letters.

5.3 Read-Only Transactions

Read-only transactions—the only read operations in Eiger—enable clients to see a consistent view of multiple keys that may be spread across many servers in the local datacenter. Eiger's algorithm guarantees low latency because it takes at most two rounds of parallel non-blocking reads in the local datacenter, plus at most one additional round of local non-blocking checks during concurrent write transactions, detailed in §5.4.4. In contrast with the read-only transactions in COPS-RT, Eiger's algorithm is based on logical time and does not use dependency graphs.

5.3.1 Read-Only Transaction Algorithm

The key insight in the algorithm is that *there exists a consistent result for every query at every logical time*. Figure 5.3 illustrates this: as operations are applied in a consistent causal order, every data location (key and column) has a consistent value at each logical time.

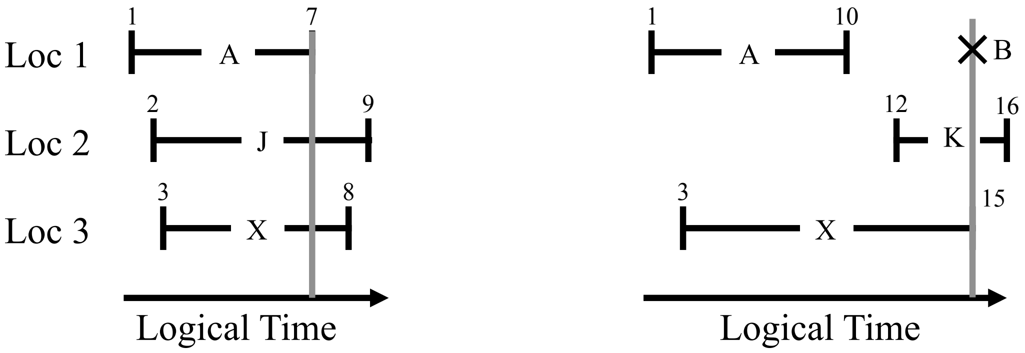
At a high level, our new read transaction algorithm marks each data location with validity metadata, and uses that metadata to determine if a first round of optimistic reads is consistent. If the first round results are not consistent, the algorithm issues a second round of reads that are guaranteed to return consistent results.

More specifically, each data location is marked with an *earliest valid time* (EVT). The EVT is set to the server's logical time when it locally applies an operation that writes a value. Thus, in an operation's accepting datacenter—the one at which the operation originated—the EVT is the same as its timestamp. In other datacenters, the EVT is later than its timestamp. In both cases, the EVT is the exact logical time when the value became visible in the local datacenter.

A server responds to a read with its currently visible value, the corresponding EVT, and its current logical time, which we call the *latest valid time* (LVT). Because this value is still visible, we know it is valid for at least the interval between the EVT and LVT. Once all first-round reads return, the client library compares their times to check for consistency. In particular, it knows all values were valid at the same logical time (i.e., correspond to a consistent snapshot) iff the maximum EVT $\leq$ the minimum LVT. In other words, the results are consistent if there is at least one logical time that falls between the EVTs and LVTs for each of the returned values. If this is the case, the client library returns these results; otherwise, it proceeds to a second round. Figure 5.4(a) shows a scenario that completes in one round.

The *effective time* of the transaction is the smallest LVT $\geq$ the maximum EVT. In a single round transaction, this is the same as the minimum LVT across all locations. It corresponds both to a logical time in which all retrieved values are consistent, as well as the current logical time (as of its response) at a server. As such, it ensures freshness—necessary in causal+ consistency so that clients always see a progressing datacenter that reflects their own updates.

We sketch a proof that read transactions return the set of results that were visible in their local datacenter at the transaction's effective time, EffT. By construction, assume a value is visible at logical time t iff val.EVT $\leq t \leq$ val.LVT. For each returned value, if it is returned from the first round, then val.EVT $\leq$ maxEVT $\leq$ EffT by definition of maxEVT and EffT, and val.LVT $\geq$ EffT because it is not being requested in the second round. Thus, val.EVT



(a) One Round Sufficient

(b) Two Rounds Needed

Figure 5.4: Examples of read-only transactions. (a) shows one with a single round, and (b) shows one with two rounds. The effective time of each transaction is shown with a gray line; this is the time requested for location 1 in the second round in (b).

$\leq$ EffT $\leq$ val.LVT, and by our assumption, the value was visible at EffT. If a result is from the second round, then it was obtained by a second-round read that explicitly returns the visible value at time EffT, described next.

5.3.2 Two-Round Read Protocol

A read transaction requires a second round if there does not exist a single logical time for which *all* values read in the first round are valid. This can only occur when there are concurrent updates being applied locally to the requested locations. The example in Figure 5.4(b) requires a second round because location 2 is updated to value K at time 12, which is not before time 10 when location 1's server returns value A.

During the second round, the client library issues `multiget_slice_by_time` requests, specifying a read at the transaction's effective time. These reads are sent only to those locations for which it does not have a valid result, i.e., their LVT is earlier than the effective time. For example, in Figure 5.4(b) a `multiget_slice_by_time` request is sent for location 1 at time 15 and returns a new value B.

Servers respond to `multiget_slice_by_time` reads with the value that was valid at the requested logical time. Because that result may be different than the currently visible one,

```
function read_only_trans(requests):
  # Send first round requests in parallel
  for r in requests
    val[r] = multiget_slice(r)

  # Calculate the maximum EVT
  maxEVT = 0
  for r in requests
    maxEVT = max(maxEVT, val[r].EVT)

  # Calculate effective time
  EffT = $\infty$
  for r in requests
    if val[r].LVT $\geq$ maxEVT
      EffT = min(EffT, val[r].LVT)

  # Send second round requests in parallel
  for r in requests
    if val[r].LVT < EffT
       val[r] = multiget_slice_by_time(r, EffT)

  # Return only the requested data
  return extract_keys_to_columns(res)
```

Figure 5.5: Pseudocode for read-only transactions.

servers sometimes must store old values for each location. Fortunately, the extent of such additional storage can be limited significantly.

5.3.3 Limiting Old Value Storage

Eiger limits the need to store old values in two ways. First, read transactions have a timeout that specifies their maximum real-time duration. If this timeout fires—which happens only when server queues grow pathologically long due to prolonged overload—the client library restarts a fresh read transaction. Thus, servers only need to store old values that have been overwritten within this timeout's duration.

Second, Eiger retains only old values that could be requested in the second round. Thus, servers store only values that are *newer* than those returned in a first round within the timeout duration. For this optimization, Eiger stores the last access time of each value.

5.3.4 Read Transactions for Linearizability

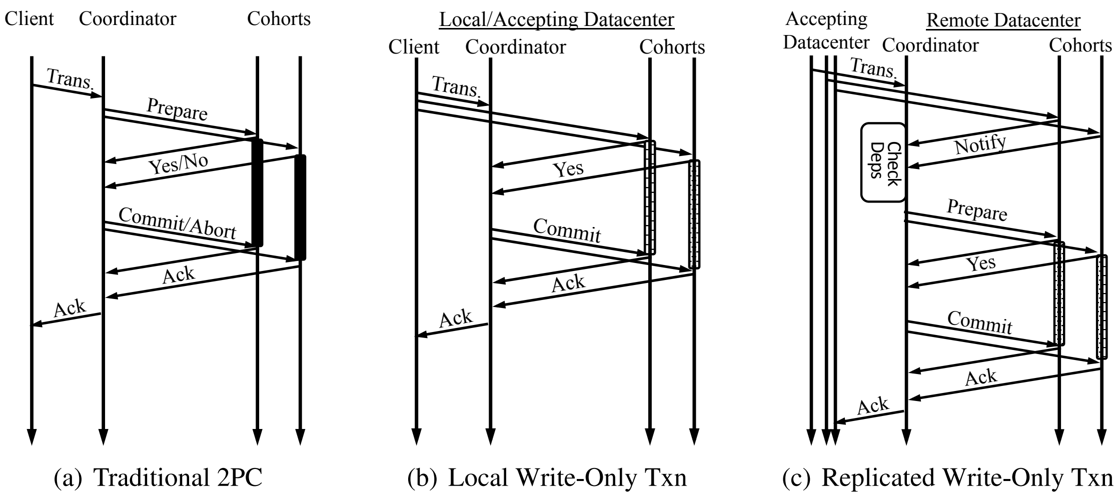
Linearizability (strong consistency) is attractive to programmers when low latency and availability are not strict requirements. Simply being linearizable, however, does not mean that a system is transactional: there may be no way to extract a mutually consistent set of values from the system, much as in our earlier example for read transactions. Linearizability is only defined on, and used with, operations that read or write a single location (originally, shared memory systems) [41].

Interestingly, our algorithm for read-only transactions works for fully linearizable systems, without modification. In Eiger, in fact, if all writes that are concurrent with a read-only transaction originated from the local datacenter, the read-only transaction provides a consistent view of that linearizable system (the local datacenter).

5.4 Write-Only Transactions

Eiger's write-only transactions allow a client to atomically write many columns spread across many keys in the local datacenter. These values also appear atomically in remote datacenters upon replication. As we will see, the algorithm guarantees low latency because it takes at most 2.5 message RTTs in the local datacenter to complete, no operations acquire locks, and all phases wait on only the previous round of messages before continuing.

Write-only transactions have many uses. When a user presses a save button, the system can ensure that all of her five profile updates appear simultaneously. Similarly, they help maintain symmetric relationships in social networks: when Alice accepts Bob's friendship request, both friend associations appear at the same time.



(a) Traditional 2PC (b) Local Write-Only Txn (c) Replicated Write-Only Txn

Figure 5.6: Message flow diagrams for traditional 2PC and write-only transaction. Solid boxes denote when cohorts block reads. Striped boxes denote when cohorts will indirect a commitment check to the coordinator.

5.4.1 Write-Only Transaction Algorithm

To execute an `atomic_mutate` request—which has identical arguments to `batch_mutate`—the client library splits the operation into one sub-request per local server across which the transaction is spread. The library randomly chooses one key in the transaction as the *coordinator key*. It then transmits each sub-request to its corresponding server, annotated with the coordinator key.

Our write transaction is a variant of two-phase commit [66], which we call *two-phase commit with positive cohorts and indirection* (2PC-PCI). 2PC-PCI operates differently depending on whether it is executing in the original (or "accepting") datacenter, or being applied in the remote datacenter after replication.

There are three differences between traditional 2PC and 2PC-PCI, as shown in Figure 5.6. First, 2PC-PCI has only positive cohorts; the coordinator always commits the transaction once it receives a vote from all cohorts.[3] Second, 2PC-PCI has a different pre-vote phase

[3] Eiger only has positive cohorts because it avoids all the normal reasons to abort (vote no): It does not have general transactions that can force each other to abort, it does not have users that can cancel operations, and it assumes that its logical servers do not fail.

that varies depending on the origin of the write transaction. In the accepting datacenter (we discuss the remote below), the client library sends each participant its sub-request directly, and this transmission serves as an implicit PREPARE message for each cohort. Third, 2PC-PCI cohorts that cannot answer a query—because they have voted but have not yet received the commit—ask the coordinator if the transaction is committed, effectively *indirecting* the request through the coordinator.

5.4.2 Local Write-Only Transactions

When a *participant* server, which is either the coordinator or a cohort, receives its transaction sub-request from the client, it prepares for the transaction by writing each included location with a special "pending" value (retaining old versions for second-round reads). It then sends a YESVOTE to the coordinator.

When the coordinator receives a YESVOTE, it updates its count of prepared keys. Once all keys are prepared, the coordinator commits the transaction. The coordinator's current logical time serves as the (global) timestamp and (local) EVT of the transaction and is included in the COMMIT message.

When a cohort receives a COMMIT, it replaces the "pending" columns with the update's real values, and ACKs the committed keys. Upon receiving all ACKs, the coordinator safely cleans up its transaction state.

5.4.3 Replicated Write-Only Transactions

Each transaction sub-request is replicated to its "equivalent" participant(s) in the remote datacenter, possibly splitting the sub-requests to match the remote key partitioning. When a cohort in a remote datacenter receives a sub-request, it sends a NOTIFY with the key count to the transaction coordinator in its datacenter. This coordinator issues any necessary `dep_checks` upon receiving its own sub-request (which contains the coordinator key). The coordinator's checks cover the entire transaction, so cohorts send no checks. Once the coordinator has received all NOTIFY messages and dep_checks responses, it sends each cohort a PREPARE, and then proceeds normally.

For reads received during the *indirection window* in which participants are uncertain about the status of a transaction, cohorts must query the coordinator for its state. To minimize the duration of this window, before preparing, the coordinator waits for (1) *all* participants to NOTIFY and (2) all dep_checks to return. This helps prevent a slow replica from causing needless indirection.

Finally, replicated write-only transactions differ in that participants do not always write pending columns. If a location's current value has a newer timestamp than that of the transaction, the validity interval for the transaction's value is empty. Thus, no read will ever return it, and it can be safely discarded. The participant continues in the transaction for simplicity, but does not need to indirect reads for this location.

5.4.4 Reads when Transactions are Pending

If a first-round read accesses a location that could be modified by a pending transaction, the server sends a special empty response that only includes a LVT (i.e., its current time). This alerts the client that it must choose an effective time for the transaction and send the server a second-round multiget_slice_by_time request.

When a server with pending transactions receives a multiget_slice_by_time request, it first traverses its old versions for each included column. If there exists a version valid at the requested time, the server returns it.

Otherwise, there are pending transactions whose *potential commit window* intersects the requested time and the server must resolve their ordering. It does so by sending a commit_check with this requested time to the transactions' coordinator(s). Each coordinator responds whether the transaction had been committed at that (past) time and, if so, its commit time.

Once a server has collected all `commit_check` responses, it updates the validity intervals of all versions of all relevant locations, up to at least the requested (effective) time. Then, it can respond to the `multiget_slice_by_time` message as normal.

The complementary nature of Eiger's transactional algorithms enables the atomicity of its writes. In particular, the single commit time for a write transaction (EVT) and the single effective time for a read transaction lead each to appear at a single logical time, while its two-phase commit ensures all-or-nothing semantics.

5.4.5 Guaranteed Low Latency

Write-only transactions can guarantee low latency because they take only 2.5 *local* message RTTs, no operations acquire locks, and all phases wait on only the previous round of messages before continuing.

Read-only transactions can still guarantee low latency for the same reasons. To see this, consider the worst case, a two-round-and-indirected read transaction. That read transaction would issue a first round of parallel reads that would immediately return. Then it would determine it needs a second round and send out a second (smaller) parallel round of reads. When those reads reached their respective servers, they might find relevant pending transactions and then would send out a `commit_check` to each's coordinators. Those coordinators then, because the requested time is in the past, can immediately respond with the status of the transaction. (The coordinator's checking of its state is causally after the first-round read that set the effective time of the transaction, and thus its clock is currently past that time.) The issuing servers could then immediately respond to the client. This transaction would require only three local RTTs—first round, second round, indirection round—as a worst case.

Chapter 6

Garbage, Faults, and Conflicts

This section describes three important aspects of COPS, COPS-RT, and Eiger: their garbage collection subsystems that reduce the amount of extra state in the system; their fault tolerant design; and their conflict detection mechanisms.

6.1 Garbage Collection Subsystem

COPS, COPS-RT, and Eiger clients store metadata; COPS-RT and Eiger servers keep multiple versions of keys; and COPS-RT servers store dependencies. Without intervention, the space footprint of the system would grow without bound as keys are updated and inserted. Our garbage collection subsystem deletes unneeded state, keeping the total system size in check. Section 7 evaluates the overhead of maintaining and transmitting this additional metadata.

Version Garbage Collection. (COPS-RT, Eiger)

What is stored: COPS-RT and Eiger store multiple versions of each key to answer `read_by_time`/`multiget_slice_by_time` requests from clients.

Why it can be cleaned: The read-only transaction algorithm limits the number of versions needed to complete a read-only transaction. Each algorithm's second round issues requests only for versions later than those returned in the first round. To enable prompt garbage collection, COPS-RT and Eiger limit the total running time of read-only transaction through a configurable parameter, *trans_time* (set to 5 seconds in our implementation). If the timeout fires, the client library will restart the read-only transaction call and satisfy the transaction with newer versions of the keys; we expect this to happen only if multiple nodes in a cluster crash.

When it can be cleaned: After a new version of a key is written, COPS-RT and Eiger only need to keep the old version around for *trans_time* plus a small delta for clock skew. After this time, no second-round read will subsequently request the old version, and the garbage collector can remove it.

Space Overhead: The space overhead is bounded by the number of old versions that can be created within the *trans_time*. This number is determined by the maximum write throughput that the node can sustain. Our COPS-RT implementation sustains 105MB/s of write traffic per node, requiring (potentially) a non-prohibitive extra 525MB of buffer space to hold old versions. This overhead is per-machine and does not grow with the cluster size or the number of datacenters.

Dependency Garbage Collection. (COPS-RT only)

What is stored: Dependencies are stored to allow read-only transactions to obtain a consistent view of the data store.

Why it can be cleaned: COPS-RT can garbage collect these dependencies once the versions associated with old dependencies are no longer needed for correctness in read-only transaction operations.

To illustrate when read-only transaction operations no longer need dependencies, consider value z_2 that depends on x_2 and y_2. A read-only transaction of x, y, and z requires that if z_2 is returned, then $x_{\geq 2}$ and $y_{\geq 2}$ must be returned as well. Causal consistency ensures that x_2 and y_2 are written before z_2 is written. Causal+ consistency's progressing property ensures that once x_2 and y_2 are written, then either they or some later version will always be returned by a read-only operation. Thus, once z_2 has been written in all datacenters and the *trans_time* has passed, any read-only transaction returning z_2 will return $x_{\geq 2}$ and $y_{\geq 2}$, and thus z_2's dependencies can be garbage collected.

When it can be cleaned: After *trans_time* seconds after a value has been committed in *all* datacenters, COPS-RT can clean a value's dependencies. (Recall that committed enforces that its dependencies have been satisfied.) COPS, COPS-RT, and Eiger can further set the value's *never-depend* flag, discussed below. To clean dependencies each remote datacenter notifies the originating datacenter when the write has committed and the timeout period has elapsed. Once all datacenters confirm, the originating datacenter cleans its own dependencies and informs the others to do likewise. To minimize bandwidth devoted to cleaning dependencies, a replica only notifies the originating datacenter if this version of a key is the newest after *trans_time* seconds; if it is not, there is no need to collect the dependencies because the entire version will be collected.

Space Overhead: Under normal operation, dependencies are garbage collected after *trans_time* plus a round-trip time. Dependencies are only collected on the most recent version of the key; older versions of keys are already garbage collected as described above.

During a partition, dependencies on the most recent versions of keys cannot be collected. This is a limitation of COPS-RT, although we expect long partitions to be rare. To illustrate why this concession is necessary for read-only transaction correctness, consider value b_2 that depends on value a_2: if b_2's dependence on a_2 is prematurely collected, some later value c_2 that causally depends on b_2—and thus on a_2—could be written without the explicit dependence on a_2. Then, if a_2, b_2, and c_2 are all replicated to a datacenter in short order, the first round of a read-only transaction could obtain a_1, an earlier version of a, with c_2, and then return these two values to the client, precisely because it did not know c_2 depends on the newer a_2.

Client Metadata Garbage Collection. (COPS, COPS-RT, Eiger)

What is Stored: The client library tracks all operations during a client session (single thread of execution) using the *actor_id* passed with all operation. In contrast to the dependency information discussed above which resides in the key-value store itself, the dependencies discussed here are part of the client metadata and are stored in the client library. In all systems, each `read` since the last `write` adds another one-hop dependency. Additionally in COPS-RT, all new values and their dependencies returned in `read_trans` operations and all `write` operations add normal dependencies. If a client session lasts for a long time, the number of dependencies attached to updates will grow large, increasing the size of the dependency metadata that our systems needs to store. The design of this garbage collection is applicable to COPS, COPS-RT, and Eiger, though we have only currently implemented it for COPS and COPS-RT.

Why it can be cleaned: As with the dependency tracking above, clients need to track dependencies only until they are guaranteed to be satisfied everywhere.

When it can be cleaned: We reduce the size of this client state (the context) in two ways. First, as noted above, once a write commits successfully to all datacenters, we flag that key version as *never-depend*, in order to indicate that clients need not express a dependence upon it. Read results include this flag, and the client library will immediately remove a *never-depend* item from the list of dependencies in the client context. Furthermore, this process is transitive: anything that a *never-depend* key depended on must have been flagged *never-depend*, so it too can be garbage collected from the context.

Second, our servers remove unnecessary dependencies from write operations. When a server receives a write, it checks each item in the dependency list and removes items with version numbers older than a *global checkpoint time*. This checkpoint time is the newest Lamport timestamp that is satisfied at *all* nodes across the entire system. The server returns this checkpoint time to the client library (e.g., in response to a write), allowing the library to clean these dependencies from the context. Because of outstanding reads, clients and servers must also wait *trans_time* seconds before they can use a new global checkpoint time.

To compute the global checkpoint time, each storage node first determines the oldest Lamport timestamp of any *pending* write in the key range for which it is primary. (In other words, it determines the timestamp of its oldest key that is not guaranteed to be satisfied at all replicas.) It then contacts its equivalent nodes in other datacenters (those nodes that handle the same key range). The nodes pair-wise exchange their minimum Lamport times, remembering the oldest observed Lamport clock of any of the replicas. At the conclusion of this step, all datacenters have the same information: each node knows the globally oldest Lamport timestamp in its key range. The nodes within a datacenter then gossip around the per-range minimums to find the minimum Lamport timestamp observed by any one of them. This periodic procedure is performed 10 times a second in our COPS implementation and has no noticeable impact on performance.

6.2 Fault Tolerance

In this section, we examine how COPS, COPS-RT, and Eiger behaves under failures, including single server failure, meta-client redirection, and entire datacenter failure.

Single server failures are common and unavoidable in practice. We guard against their failure with the construction of logical servers from multiple physical servers. For instance, a logical server implemented with a three-server Paxos group can withstand the failure of one of its constituent servers. Like any system built on underlying components, our systems inherit the failure modes of their underlying building blocks. In particular, if a logical server assumes no more than f physical machines fail, we must assume that within a single logical server no more than f physical machines fail.

Meta-clients that are the clients of our systems' clients (i.e., web browsers that have connections to front-end web tier machines) will sometimes be directed to a different datacenter. For instance, a redirection may occur when there is a change in the DNS resolution policy of a service. When a redirection occurs during the middle of an active connection, we expect service providers to detect it using cookies and then redirect clients to their original datacenter (e.g., using HTTP redirects or triangle routing). When a client is not actively using the service, however, policy changes that reassign it to a new datacenter can proceed without complication.

Datacenter failure can either be transient (e.g., network or power cables are cut) or permanent (e.g., datacenter is physically destroyed by an earthquake). Permanent failures will result in data loss for data that was accepted and acknowledged but not yet replicated to any other datacenter. The colocation of clients inside the datacenter, however, will reduce the amount of externally visible data loss. Only data that is not yet replicated to another datacenter, but has been acknowledged to both Eiger's clients and meta-clients (e.g., when the browser receives an Ajax response indicating a status update was posted) will be visibly lost. Transient datacenter failure will not result in data loss.

Both transient and permanent datacenter failures will cause meta-clients to reconnect to different datacenters. After some configured timeout, we expect service providers to stop trying to redirect those meta-clients to their original datacenters and to connect them to a new datacenter with an empty context. This could result in those meta-clients effectively moving backwards in time. It would also result in the loss of causal links between the data they observed in their original datacenter and their new writes issued to their new datacenter. We expect that transient datacenter failure will be rare (no ill effects), transient failure that lasts long enough for meta-clients to use a different datacenter even rarer (causality loss), and permanent failure even rarer still (data loss).

COPS-RT is less resilient to datacenter failure than COPS and Eiger. Specifically, in COPS-RT, dependency garbage collection cannot continue in the face of a datacenter failure, until either the partition is healed or the system is reconfigured to exclude the failed datacenter. Without dependency garbage collection, the number of dependencies in

COPS-RT will continue to grow. This increases the metadata associated with each operation and the number of dependency checks needed, thus continually lowering the goodput of COPS-RT. COPS and Eiger do not suffer from this weakness.

6.3 Conflict Detection

Conflicts occur when there are two “simultaneous” (i.e., not in the same context/thread of execution) writes to a given key. The systems by default avoid conflict detection using a last-writer-wins strategy. The “last” write is determined by comparing timestamps, and allows us to avoid conflict detection for increased simplicity and efficiency. We believe this behavior is useful for many applications. There are other applications, however, that become simpler to reason about and program with a more explicit conflict-detection scheme. For these applications, COPS can be configured to detect conflicting operations and then invoke some application-specific convergent conflict handler.[1]

COPS with conflict detection, which we refer to as COPS-CD, adds three new components to the system. First, all write operations carry with them *previous version* metadata, which indicates the most recent previous version of the key that was visible at the local cluster at the time of the write (this previous version may be null). Second, all write operations now have an implicit dependency on that previous version, which ensures that a new version will only be written after its previous version. This implicit dependency entails an additional `dep_check` operation, though this has low overhead and always executes on the local machine. Third, COPS-CD has an application-specified convergent conflict handler that is invoked when a conflict is detected.

COPS-CD follows a simple procedure to determine if a write operation *new* to a key (with previous version *prev*) is in conflict with the key’s current visible version *curr*:

$prev \neq curr$ if and only if *new* and *curr* conflict.

[1] Eiger does not currently provide conflict detection and adding it is an interesting direction of future work. In particular, detecting and resolving intersecting write transactions will be challenging.

We present a proof sketch here. In the forward direction, we know that *prev* must be written before *new*, $prev \neq curr$, and that for *curr* to be visible instead of *prev*, we must have $curr > prev$ by the progressing property of causal+. But because *prev* is the most recent causally previous version of *new*, we can conclude $curr \not\rightsquigarrow new$. Further, because *curr* was written before *new*, it cannot be causally after it, so $new \not\rightsquigarrow curr$ and thus they conflict. In the reverse direction, if *new* and *curr* conflict, then $curr \not\rightsquigarrow new$. By definition, $prev \rightsquigarrow new$, and thus $curr \neq prev$.

Chapter 7

Evaluation

This section presents an evaluation of COPS and COPS-RT, and a separate evaluation of Eiger. Because COPS and Eiger provide different data models and are implemented in different languages a direct empirical comparison between them is not meaningful. In Section 7.4 we do, however, analytically compare COPS and Eiger.

7.1 Experimental Setup

All experiments, excluding Eiger's scaling experiment, use the shared VICCI testbed [60, 75], which provides users with Linux VServer instances. Each physical machine has 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, 3x1TB HDDs, and 2x1GigE network ports.

For experiments with COPS and COPS-RT in Section 7.2 we partition the cluster into multiple logical "datacenters" as necessary. Microbenchmarks use 1 machine/cluster, dynamic workload experiments use 4 machines/cluster, scaling experiments use up to 16 machines/cluster. We retain full bandwidth between the nodes in different datacenters to reflect the high-bandwidth backbone that often exists between them. Our later experiment with Eiger on VICCI in a wide-area setup confirm wide-area bandwidth is not a bottleneck.

For Eiger, all experiments are between multiple VICCI sites. The latency microbenchmark uses a minimal wide-area setup with a cluster of 2 machines at the Princeton, Stanford, and University of Washington (UW) VICCI sites. All other experiments use 8-machine clusters in Stanford and UW and an additional 8 machines in Stanford as clients. These clients fully load their local cluster, which replicates its data to the other cluster.

The inter-site latencies were 88ms between Princeton and Stanford, 84ms between Princeton and UW, and 20ms between Stanford and UW. Inter-site bandwidth was not a limiting factor.

Every datapoint in the evaluation represents the median of 5+ trials. Latency microbenchmark trials are 30s, while all other trials are 60s. We elide the first and last quarter of each trial to avoid experimental artifacts.

7.2 COPS

We evaluate COPS and COPS-RT with microbenchmarks that establish baseline system latency and throughput, a sensitivity analysis that explores the impact of different parameters that characterize a dynamic workload, and larger end-to-end experiments that show scalable causal+ consistency.

7.2.1 Implementation

COPS is approximately 13,000 lines of C++ code. It is built on top of FAWN-KV [6, 31] (~8500 LOC), which provides linearizable key-value storage within a local cluster. COPS uses Apache Thrift [8] for communication between all system components and Google's Snappy [67] for compressing dependency lists. Our current prototype implements all features described in the paper, excluding conflict detection.

We compare three systems: LOG, COPS, and COPS-RT. LOG uses the COPS codebase but excludes all dependency tracking, making it simulate previous work that uses log exchange to establish causal consistency (e.g., Bayou [59] and PRACTI [12]). Table 7.1 summarizes these three systems.

System	**Causal+**	**Scalable**	**Read-only Transactions**
LOG	Yes	No	No
COPS	Yes	Yes	No
COPS-RT	Yes	Yes	Yes

Table 7.1: Summary of three systems under comparison.

System	**Operation**	**Latency (ms)**			**Throughput**
		50%	**99%**	**99.9%**	**(Kops/s)**
Thrift	ping	0.26	3.62	12.25	60
COPS	read_by_version	0.37	3.08	11.29	52
COPS-RT	read_by_version	0.38	3.14	9.52	52
COPS	write_after (1)	0.57	6.91	11.37	30
COPS-RT	write_after (1)	0.91	5.37	7.37	24
COPS-RT	write_after (130)	1.03	7.45	11.54	20

Table 7.2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. write_after(x) includes metadata for x dependencies.

7.2.2 Microbenchmarks

We first evaluate the performance characteristics for COPS and COPS-RT in a simple setting: two datacenters, one server per datacenter, and one colocated client machine. The client sends read and write requests to its local server, attempting to saturate the system. The requests are spread over 2^{18} keys and have 1B values—we use 1B values for consistency with later experiments, where small values are the worst case for COPS (see Figure 7.4(c)). Table 7.2 shows the median, 99%, and 99.9% latencies and throughput.

The design decision in COPS to handle client operations locally yields low latency for all operations. The latencies for `read_by_time` operations in COPS and COPS-RT are similar to an end-to-end RPC ping using Thrift. The latencies for `write_after` operations are slightly higher because they are more computationally expensive; they need to update metadata and write values. Nevertheless, the median latency for `write_after` operations, even those with up to 130 dependencies, is around 1 ms.

System throughput follows a similar pattern. `read_by_time` operations achieve high throughput, similar to that of Thrift ping operations (52 vs. 60 Kops/s). A COPS server can process `write_after` operations at 30 Kops/s (such operations are more computationally expensive than reads), while COPS-RT achieves 20% lower throughput when `write_after` operations have 1 dependency (due to the cost of garbage collecting old versions). As the number of dependencies in COPS-RT `write_after` operations increases, throughput drops slightly due to the greater size of metadata in each operation (each dependency is ~12B).

7.2.3 Dynamic Workloads

We model a dynamic workload with interacting clients accessing the COPS system as follows. We set up two datacenters of S servers each and colocate S client machines in one of the two datacenters. The clients access storage servers in the local datacenter, which replicates `write_after` operations to the remote datacenter. We report the sustainable throughput in our experiments, which is the maximum throughput that both datacenters can handle. In most cases, COPS becomes CPU-bound at the local datacenter, and that COPS-RT becomes CPU-bound at the remote one.

To better stress the system and more accurately depict real operation, each client machine emulates multiple logical COPS clients. Each time a client performs an operation, it randomly executes a read or write, according to a specified write:read ratio. All operations in a given experiment use fixed-size values.

The key for each operation is selected to control the amount of dependence between operations (i.e., from fully isolated to fully intermixed). Specifically, given N clients, the full keyspace consists of N keygroups, $R_1 \ldots R_N$, one per client. Each keygroup contains K keys, which are randomly distributed (i.e., they do not all reside on the same server). When clients issue operations, they select keys as follows. First, they pick a keygroup by sampling from a normal distribution defined over the N keygroups, where each keygroup has width 1. Then, they select a key within that keygroup uniformly at random. The result

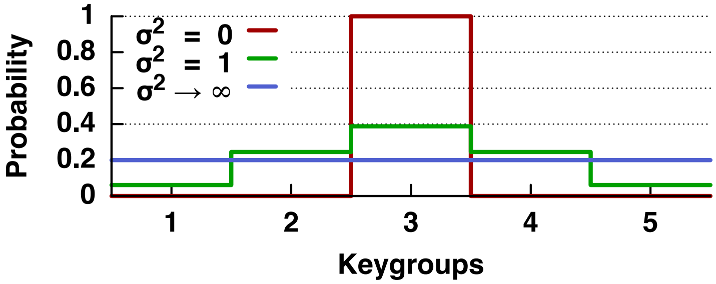


Figure 7.1: In our experiments, clients choose keys to access by first selecting a keygroup according to some normal distribution, then randomly selecting a key within that group according to a uniform distribution. Figure shows such a stepped normal distribution for differing variances for client #3 (of 5).

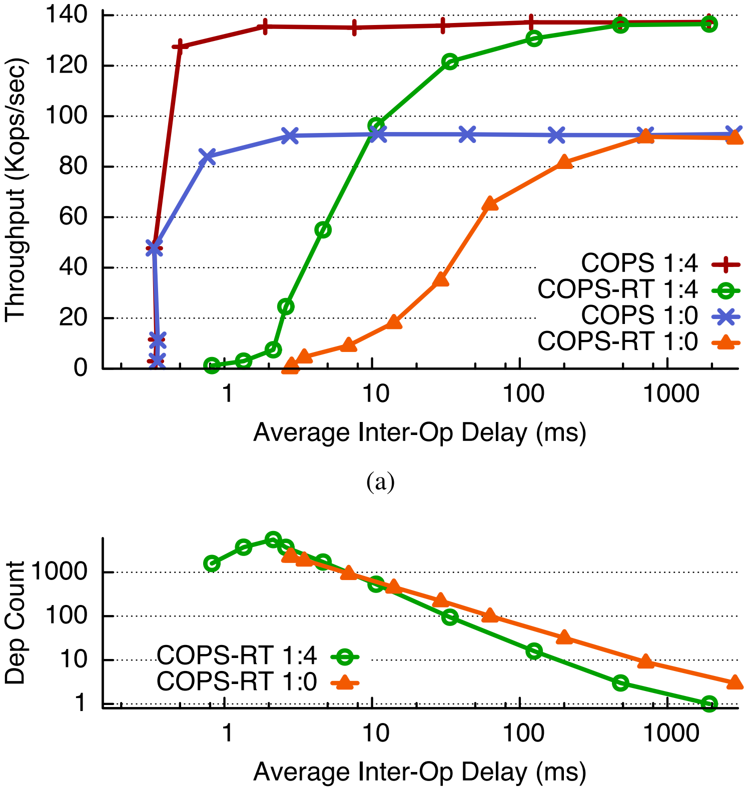
Parameter	**Range**	**Default**
clients/server	1-2^{18}	1024
keys/keygroup	1-2048	512
write:read ratio	64:1-1:64	1:1 or 1:4
variance	1/64-512	1
value size	1B-16KB	1B

Table 7.3: COPS dynamic workload generator parameters. Range is the space covered in the experiments.

is a distribution over keys with equal likelihood for keys within the same keygroup, and possibly varying likelihood across groups.

Figure 7.1 illustrates an example, showing the keygroup distribution for client #3 (of 5 total) for variances of 0, 1, and the limit approaching ∞. When the variance is 0, a client will restrict its accesses to its “own” keygroup and never interact with other clients. In contrast, when the variance $\rightarrow \infty$, client accesses are distributed uniformly at random over all keys, leading to maximal inter-dependencies between `write_after` operations.

The parameters of the dynamic workload experiments are those give in Table 7.3 unless otherwise specified. As the state space of all possible combinations of these variables is large, the following experiments explore parameters individually.



(a)

(b)

Figure 7.2: Maximum throughput and the resulting average dependency size of COPS and COPS-RT for a given inter-write delay between consecutive operations by the same logical client. The legend gives the write:read ratio (i.e., 1:0 or 1:4).

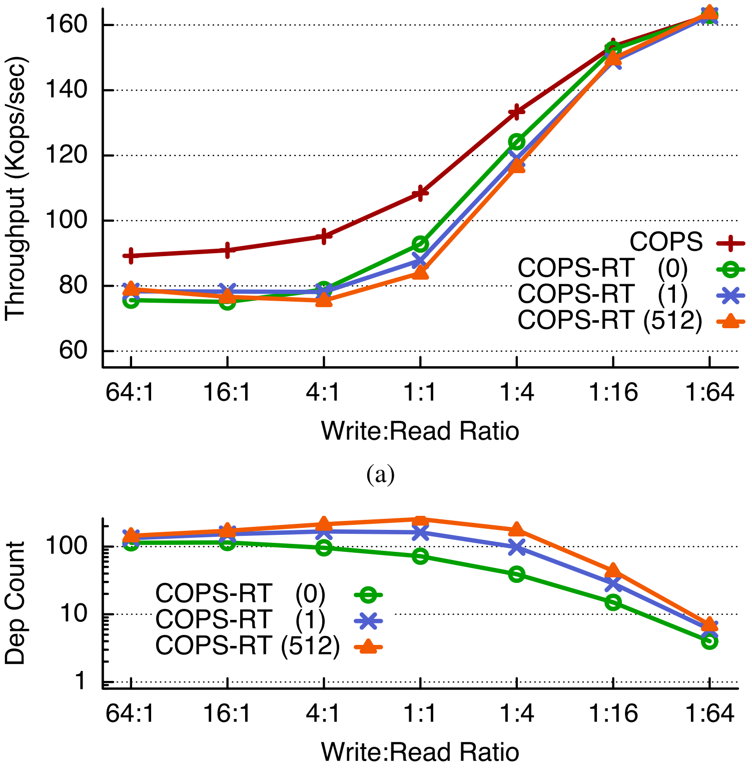
Clients Per Server. We first characterize the system throughput as a function of increasing delay between client operations (for two different write:read ratios).[1] Figure 7.2(a) shows that when the inter-operation delay is low, COPS significantly outperforms COPS-RT. Conversely, when the inter-operation delay approaches several hundred milliseconds, the maximum throughputs of COPS and COPS-RT converge. Figure 7.2(b) helps explain this behavior: as the inter-operation delay increases, the number of dependencies per operation *decreases* because of the ongoing garbage collection.

[1] For these experiments, we do not directly control the inter-operation delay. Rather, we increase the number of logical clients running on each of the client machines from 1 to 2^{18}; given a fixed-size thread pool for clients in our test framework, each logical client gets scheduled more infrequently. As each client makes one request before yielding, this leads to higher average inter-op delay (calculated simply as $\frac{\text{throughput}}{\text{\# of clients}}$). Our default setting of 1024 clients/server yields an average inter-op delay of 29 ms for COPS-RT with a 1:0 write:read ratio, 11ms for COPS with 1:0, 11ms for COPS-RT with 1:4, and 8ms for COPS with 1:4.

To understand this relationship, consider the following example. If the global-checkpoint-time is 6 seconds behind the current time and a logical client is performing 100 writes/sec (in an all-write workload), each write will have $100 \cdot 6 = 600$ dependencies. Figure 7.2(b) illustrates this relationship. While COPS will store only the single nearest dependency (not shown), COPS-RT must track all dependencies that have not been garbage collected. These additional dependencies explain the performance of COPS-RT: when the inter-write time is small, there are a large number of dependencies that need to be propagated with each value, and thus each operation is more expensive.

The global-checkpoint-time typically lags ~6 seconds behind the current time because it includes both the *trans_time* delay (per subsection 6.1) and the time needed to gossip checkpoints around their local datacenter (nodes gossip once every 100ms). Recall that an agreed-upon *trans_time* delay is needed to ensure that currently executing `read_trans` operations can complete, while storage nodes use gossiping to determine the oldest uncommitted operation (and thus the latest timestamp for which dependencies can be garbage collected). Notably, round-trip-time latency *between* datacenters is only a small component of the lag, and thus performance is not significantly affected by RTT (e.g., a 70ms wide-area RTT is about 1% of a 6s lag for the global-checkpoint-time).

Write:Read Ratio. We next evaluate system performance under varying write:read ratios and key-access distributions. Figure 7.3(a) shows the throughput of COPS and COPS-RT for write:read ratios from 64:1 to 1:64 and three different distribution variances. We observe that throughput increases for read-heavier workloads (write:read ratios < 1), and that COPS-RT becomes competitive with COPS for read-mostly workloads. While the performance of COPS is identical under different variances, the throughput of COPS-RT is affected by variance. We explain both behaviors by characterizing the relationship between write:read ratio and the number of dependencies (Figure 7.3(b)); fewer dependencies translates to less metadata that needs to be propagated and thus higher throughput.



(a)

(b)

Figure 7.3: Maximum throughput and the resulting average dependency size of COPS and COPS-RT for a given write:read ratio. The legend gives the variance (i.e., 0, 1, or 512).

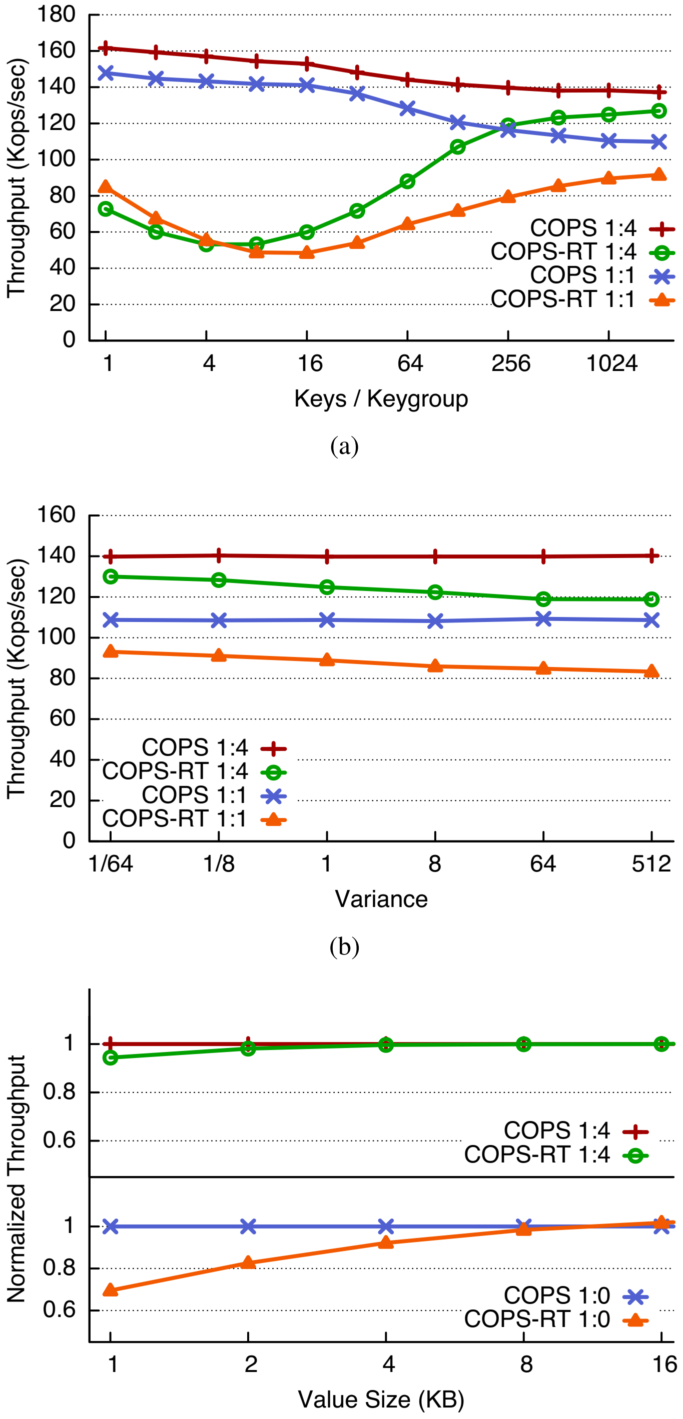
When different clients access the same keys (variance > 0), we observe two distinct phases in Figure 7.3(b). First, as the write:read ratio decreases from 64:1 to 1:1, the number of dependencies *increases*. This increase occurs because each read operation increases the likelihood a client will inherit new dependencies by reading a value that has been recently written by another client. For instance, if client$_1$ writes a value v_1 with dependencies d and client$_2$ reads that value, then client$_2$'s future write will have dependencies on both v_1 and d. Second, as the write:read ratio then decreases from 1:1 to 1:64, the number of dependencies *decreases* for two reasons: (i) each client is executing fewer write operations and thus each value depends on fewer values previously written by this client; and (ii) because there are fewer write operations, more of the keys have a value that is older than the global-checkpoint-time, and thus reading them introduces no additional dependencies.

When clients access independent keys (variance = 0), the number of dependencies is strictly decreasing with the write:read ratio. This result is expected because each client accesses only values in its own keygroup that it previously wrote and already has a dependency on. Thus, no read causes a client to inherit new dependencies.

The average dependency count for COPS (not shown) is always low, from 1 to 4 dependencies, because COPS needs to track only the one-hop (instead of all) dependencies.

Keys Per Keygroup. Figure 7.4(a) shows the effect of keygroup size on the throughput of COPS and COPS-RT. Recall that clients distribute their requests uniformly over keys in their selected keygroup. The behavior of COPS-RT is nuanced; we explain its varying throughput by considering the likelihood that a read operation will inherit new dependencies, which in turn reduces throughput. With the default variance of 1 and a low number of keys/keygroup, most clients access only a small number of keys. Once a value is retrieved and its dependencies inherited, subsequent reads on that same value do not cause a client to inherit any new dependencies. As the number of keys/keygroup begins to increase, however, clients are less likely to read the same value repeatedly, and they begin inheriting additional dependencies. As this number continues to rise, however, garbage collection can begin to have an effect: fewer reads retrieve a value that was recently written by another client (e.g., after the global checkpoint time), and thus fewer reads return new dependencies. The bowed shape of COPS-RT's performance is likely due to these two contrasting effects.

Variance. Figure 7.4(b) examines the effect of variance on system performance. As noted earlier, the throughput of COPS is unaffected by different variances: read operations in COPS never inherit extra dependencies, as the returned value is always "nearer," by definition. COPS-RT has an increased chance of inheriting dependencies as variance increases, however, which results in decreased throughput.



(a)

(b)

(c)

Figure 7.4: Maximum system throughput (using write:read ratios of 1:4, 1:1, or 1:0) for varied keys/keygroup, variances, and value sizes.

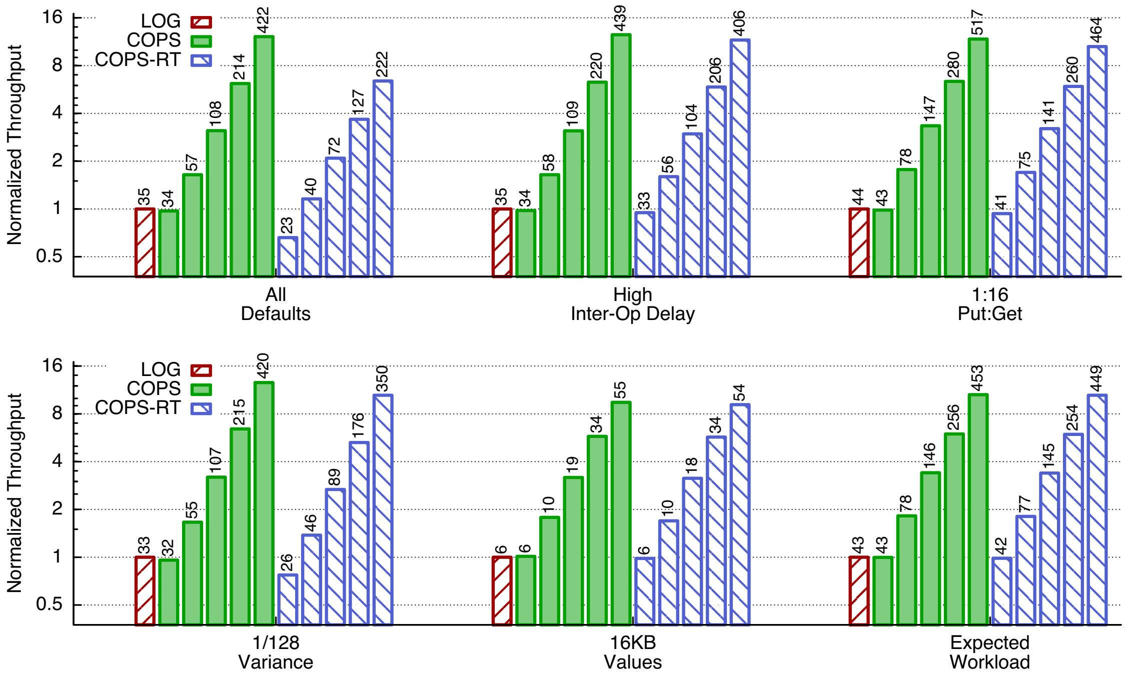


Figure 7.5: Throughput for LOG with 1 server/datacenter, and COPS and COPS-RT with 1, 2, 4, 8, and 16 servers/datacenter, for a variety of scenarios. Throughput is normalized against LOG for each scenario; raw throughput (in Kops/s) is given above each bar.

Value Size. Finally, Figure 7.4(c) shows the effect of value size on system performance. In this experiment, we normalize the systems' maximum throughput against that of COPS (the COPS line at exactly 1.0 is shown only for comparison). As the size of values increases, the relative throughput of COPS-RT approaches that of COPS.

We attribute this to two reasons. First, the relative cost of processing dependencies (which are of fixed size) decreases compared to that of processing the actual values. Second, as processing time per operation increases, the inter-operation delay correspondingly increases, which in turn leads to fewer dependencies.

7.2.4 Scaling

To evaluate the scalability of COPS and COPS-RT, we compare them to LOG. LOG mimics systems based on log serialization and exchange, which can only provide causal+ consistency with single node replicas. Our implementation of LOG uses the COPS code, but excludes dependency tracking.

Figure 7.5 shows the throughput of COPS and COPS-RT (running on 1, 2, 4, 8, or 16 servers/datacenter) normalized against LOG (running on 1 server/datacenter). Unless specified otherwise, all experiments use the default settings given in subsection 7.2.3, including a write:read ratio of 1:1. In all experiments, COPS running on a single server/datacenter achieves performance almost identical to LOG. (After all, compared to LOG, COPS needs to track only a small number of dependencies, typically ≤ 4, and any `dep_check` operations in the remote datacenter can be executed as local function calls.) More importantly, we see that COPS and COPS-RT scale well in all scenarios: as we double the number of servers per datacenter, throughput almost doubles.

In the experiment with all default settings, COPS and COPS-RT scale well relative to themselves, although COPS-RT's throughput is only about two-thirds that of COPS. These results demonstrate that the default parameters were chosen to provide a non-ideal workload for the system. However, under a number of different conditions—and, indeed, a workload more common to Internet services—the performance of COPS and COPS-RT is almost identical.

As one example, the relative throughput of COPS-RT is close to that of COPS when the inter-operation delay is high (achieved by hosting 32K clients per server, as opposed to the default 1024 clients; see Footnote 1). Similarly, a more read-heavy workload (write:read ratio of 1:16 vs. 1:1), a smaller variance in clients' access distributions ($1/128$ vs. 1), or larger-sized values (16KB vs. 1B)—controlling for all other parameters—all have the similar effect: the throughput of COPS-RT becomes comparable to that of COPS.

Finally, for the "expected workload" experiment, we set the parameters closer to what we might encounter in an Internet service such as social networking. Compared to the default, this workload has a higher inter-operation delay (32K clients/server), larger values

(1KB), and a read-heavy distribution (1:16 ratio). Under these settings, the throughput of COPS and COPS-RT are very comparable, and both scale well with the number of servers.

7.3 Eiger

This evaluation explores the overhead of Eiger's stronger semantics compared to eventually-consistent Cassandra and shows that Eiger scales to large clusters.

7.3.1 Implementation

Our Eiger prototype implements everything described in Chapter 5 as 5000 lines of Java added to and modifying the existing 75000 LOC in Cassandra 1.1 [21, 46]. All of Eiger's reads are transactional. We use Cassandra configured for wide-area eventual consistency as a baseline for comparison. In each local cluster, both Eiger and Cassandra use consistent hashing to map each key to a single server, and thus trivially provide linearizability.

In unmodified Cassandra, for a single logical request, the client sends all of its sub-requests to a single server. This server splits `batch_mutate` and `multiget_slice` operations from the client that span multiple servers, sends them to the appropriate server, and re-assembles the responses for the client. In Eiger, the client library handles this splitting, routing, and re-assembly directly, allowing Eiger to save a local RTT in latency and potentially many messages between servers. With this change, Eiger outperforms unmodified Cassandra in many settings. Therefore, to make our comparison to Cassandra fair, we implemented an analogous client library that handles the splitting, routing, and re-assembly for Cassandra. The results below use this optimization.

7.3.2 Eiger Overheads

We first examine the overhead of Eiger's causal consistency, read-only transactions, and write-only transactions. This section explains why each potential source of overhead does not significantly impair throughput, latency, or storage; the next sections confirm empirically.

Causal Consistency Overheads. Write operations carry *dependency metadata*. Its impact on throughput and latency is low because each dependency is 16 bytes[2]; the number of dependencies attached to a write is limited to its small set of one-hop dependencies; and writes are typically less frequent. Dependencies have no storage cost because they are not stored at the server.

Dependency check operations are issued in remote datacenters upon receiving a replicated write. Limiting these checks to the write's one-hop dependencies minimizes throughput degradation. They do not affect client-perceived latency, occuring only during asynchronous replication, nor do they add storage overhead.

Read-only Transaction Overheads. *Validity-interval metadata* is stored on servers and returned to clients with read operations. Its effect is similarly small: only the 8B EVT (earliest valid time) is stored, and the 16B of metadata returned to the client is tiny compared to typical key/column/value sets.

If *second-round reads* were always needed, they would roughly double latency and halve throughput. Fortunately, they occur only when there are concurrent writes to the requested columns in the local datacenter, which is rare given the short duration of reads and writes.

Extra-version storage is needed at servers to handle second-round reads. It has no impact on throughput or latency, and its storage footprint is small because we aggressively limit the number of old versions (see §5.3.3).

[2] Dependencies in Eiger are currently uncompressed, they would shrink to 12B if we used compression like that in COPS.

Write-only Transaction Overheads. Write transactions *write columns twice*: once to mark them pending and once to write the true value. This accounts for about half of the moderate overhead of write transactions, evaluated in §7.3.4. When only some writes are transactional and when the writes are a minority of system operations (as found in prior studies [9, 33]), this overhead has a small effect on overall throughput. The second write overwrites the first, consuming no space.

Many *2PC-PCI messages* are needed for the write-only algorithm. These messages add 1.5 local RTTs to latency, but have little effect on throughput: the messages are small and can be handled in parallel with other steps in different write transactions.

Indirected second-round reads add an extra local RTT to latency and reduce read throughput vs. normal second-round reads. They affect throughput minimally, however, because they occur rarely: only when the second-round read arrives when there is a not-yet-committed write-only transaction on an overlapping set of columns that prepared before the read-only transaction's effective time.

7.3.3 Latency Microbenchmark

Eiger always satisfies client operations within a local datacenter and thus, fundamentally, is low-latency. To demonstrate this, verify our implementation, and compare with strongly-consistent systems, we ran an experiment to compare the latency of read and write operations in Eiger vs. three Cassandra configurations: eventual (R=1, W=1), strong-A (R=3, W=1), and strong-B (R=2, W=2), where R and W indicate the number of datacenters involved in reads and writes.[3]

The experiments were run from UW with a single client thread to isolate latency differences. Table 7.4 reports the median, 90%, 95%, and 99% latencies from operations on a single 1B column. For comparison, two 1B columns, stored on different servers, were also updated together as part of transactional and non-transactional "Eiger (2)" write operations.

[3] Cassandra single-key writes are not atomic across different nodes, so its strong consistency requires read repair (write-back) and $R>N/2$.

	Latency (ms)			
	50%	**90%**	**95%**	**99%**
Reads				
Cassandra-Eventual	0.38	0.56	0.61	1.13
Eiger 1 Round	0.47	0.67	0.70	1.27
Eiger 2 Round	0.68	0.94	1.04	1.85
Eiger Indirected	0.78	1.11	1.18	2.28
Cassandra-Strong-A	85.21	85.72	85.96	86.77
Cassandra-Strong-B	21.89	22.28	22.39	22.92
Writes				
Cassandra-Eventual Cassandra-Strong-A	0.42	0.63	0.91	1.67
Eiger Normal	0.45	0.67	0.75	1.92
Eiger Normal (2)	0.51	0.79	1.38	4.05
Eiger Transaction (2)	0.73	2.28	2.94	4.39
Cassandra-Strong-B	21.65	21.85	21.93	22.29

Table 7.4: Latency micro-benchmarks.

All reads in Eiger—one-round, two-round, and worst-case two-round-and-indirected reads—have median latencies under 1ms and 99% latencies under 2.5ms. `atomic_mutate` operations are slightly slower than `batch_mutate` operations, but still have median latency under 1ms and 99% under 5ms. Cassandra's strongly consistent operations fared much worse. Configuration "A" achieved fast writes, but reads had to access all datacenters (including the ~84ms RTT between UW and Princeton); "B" suffered wide-area latency for both reads and writes (as the second datacenter needed for a quorum involved a ~20ms RTT between UW and Stanford).

7.3.4 Write Transaction Cost

Figure 7.6 shows the throughput of write-only transactions, and Cassandra's non-atomic batch mutates, when the keys they touch are spread across 1 to 8 servers. The experiment used the default parameter settings from Table 7.5 with 100% writes and 100% write transactions.

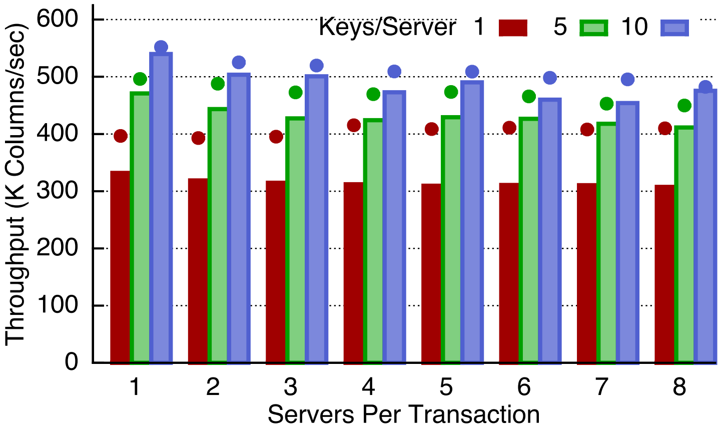


Figure 7.6: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.

Parameter	**Range**	**Default**	**Facebook**		
			50%	**90%**	**99%**
Value Size (B)	1-4K	128	16	32	4K
Cols/Key for Reads	1-32	5	1	2	128
Cols/Key for Writes	1-32	5	1	2	128
Keys/Read	1-32	5	1	16	128
Keys/Write	1-32	5	1		
Write Fraction	0-1.0	.1	.002		
Write Txn Fraction	0-1.0	.5	0 or 1.0		
Read Txn Fraction	1.0	1.0	1.0		

Table 7.5: Dynamic workload generator parameters. Range is the space covered in the experiments; Facebook describes the distribution for that workload.

Eiger's throughput remains competitive with batch mutates as the transaction is spread across more servers. Additional servers only increase 2PC-PCI costs, which account for less than 10% of Eiger's overhead. About half of the overhead of write-only transactions comes from double-writing columns; most of the remainder is due to extra metadata. Both absolute and Cassandra-relative throughput increase with the number of keys written per server, as the coordination overhead remains independent of the number of columns.

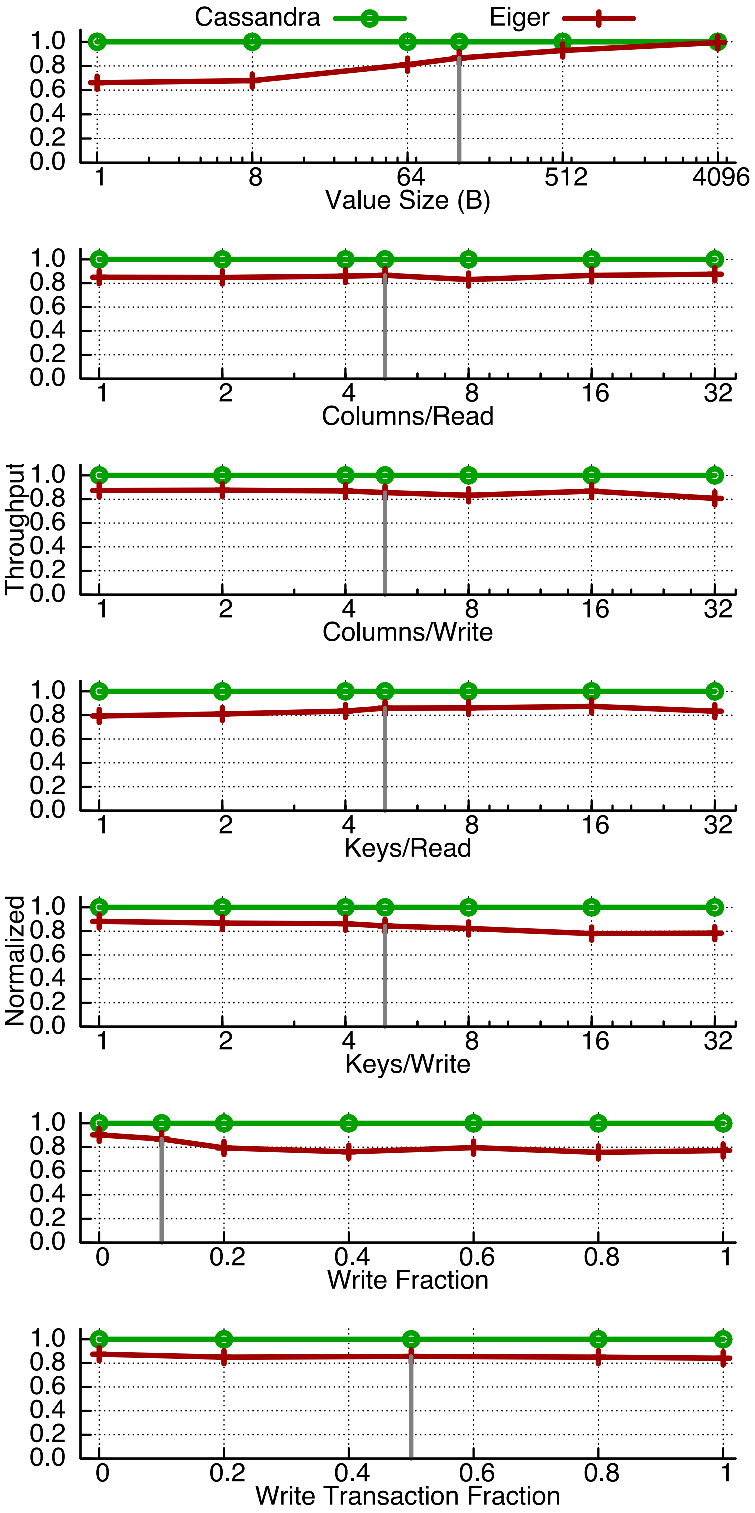


Figure 7.7: Results from exploring our dynamic-workload generator's parameter space. Each experiment varies one parameter while keeping all others at their default value (indicated by the vertical line). Eiger's throughput is normalized against eventually-consistent Cassandra.

	Ops/sec	**Keys/sec**	**Columns/sec**
Cassandra	23,657	94,502	498,239
Eiger	22,088	88,238	466,844
Eiger All Txns	22,891	91,439	480,904
Max Overhead	6.6%	6.6%	6.3%

Table 7.6: Throughput for the Facebook workload.

7.3.5 Dynamic Workloads

We created a dynamic workload generator to explore the space of possible workloads. Table 7.5 shows the range and default value of the generator's parameters. The results from varying each parameter while the others remain at their defaults are shown in Figure 7.7.

Eiger's dynamic workload generator varies from the one in COPS in one notable way: it does not have clients select keys from keygroups according to a normal distribution with some set variance. This part of the generator did not affect COPS, as we saw in Figure 7.4(b), and because Eiger, like COPS, only tracks one-hop dependencies this would not affect its performance.

We give a brief review of these results. Overhead decreases with increasing value size, because metadata represents a smaller portion of message size. Overhead is relatively constant with increases in the columns/read, columns/write, keys/read, and keys/write ratios because while the amount of metadata increases, it remains in proportion to message size. Higher fractions of write transactions (within an overall 10% write workload) do not increase overhead.

The main result of this experiment is that Eiger's throughput is overall competitive with the eventually-consistent Cassandra baseline. With the default parameters, its overhead is 15%. When they are varied, its overhead ranges from 0.5% to 25%.

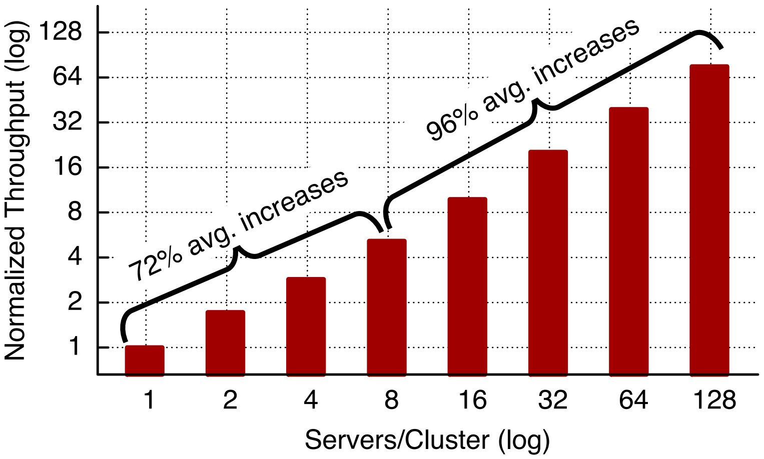


Figure 7.8: Normalized throughput of N-server clusters for the Facebook TAO workload. Bars are normalized against the 1-server cluster.

7.3.6 Facebook Workload

For one realistic view of Eiger's overhead, we parameterized a synthetic workload based upon Facebook's production TAO system [69]. Parameters for value sizes, columns/key, and keys/operation are chosen from discrete distributions measured by the TAO team. We show results with a 0% write transaction fraction (the actual workload, because TAO lacks transactions), and with 100% write transactions. Table 7.5 shows the heavy-tailed distributions' 50th, 90th, and 99th percentiles.

Table 7.6 shows that the throughput for Eiger is within 7% of eventually-consistent Cassandra. The results for 0% and 100% write transactions are effectively identical because writes are such a small part of the workload. For this real-world workload, Eiger's causal consistency and stronger semantics do not impose significant overhead.

7.3.7 Scaling

To demonstrate the scalability of Eiger we ran the Facebook TAO workload on N client machines that are fully loading an N-server cluster that is replicating writes to another N-server cluster, i.e., the N=128 experiment involves 384 machines. This experiment was run on PRObE's Kodiak testbed [62], which provides an Emulab [76] with exclusive access to hundreds of machines. Each machine has 2 AMD Opteron 252 CPUS, 8GM RAM, and an InfiniBand high-speed interface. The bottleneck in this experiment is server CPU.

Figure 7.8 shows the throughput for Eiger as we scale *N* from 1 to 128 servers/cluster. The bars show throughput normalized against the throughput of the 1-server cluster. Eiger scales as the number of servers increases, though this scaling is not linear from 1 to 8 servers/cluster. The 1-server cluster benefits from batching; all operations that involve multiple keys are executed on a single machine. Larger clusters distribute these multi-key operations over multiple servers and thus lose batching. As clusters increase in size they continue to lose batching until every key in an operation is resident on a different server. This mainly affects scaling from 1 to 8 servers/cluster with a 72% average increase per doubling of servers and we see almost perfect linear scaling from 8 to 128 servers/cluster with a 96% average increase per doubling of servers. The inflection point around 8 servers/cluster is a result of the workload, which has 73% of operations involve 4 or fewer keys and 83% of operations involve 8 or fewer keys.

7.4 Analytically Comparing COPS and Eiger

COPS and Eiger provide different data models and are implemented in different languages, so a direct empirical comparison is not meaningful. We can, however, intuit how Eiger's algorithms perform in the COPS setting.

Both COPS and Eiger achieve low latency around 1ms. Second-round reads would occur in COPS and Eiger equally often, because both are triggered by the same scenario: concurrent writes in the local datacenter to the same keys. Eiger experiences some additional latency when second-round reads are indirected, but this is rare (and the total latency remains low). Write-only transactions in Eiger would have higher latency than their non-atomic counterparts in COPS, but we have also shown their latency to be very low.

Beyond having write transactions, which COPS does not, the most significant difference between Eiger and COPS is the efficiency of Eiger's read-only transactions. COPS-RT's read-only transactions add significant dependency-tracking overhead vs. the COPS baseline under certain conditions as shown in Section 7.2.3. In contrast, by tracking only one-hop dependencies, Eiger avoids the metadata explosion that COPS' read-only transactions can suffer. We expect that Eiger's read transactions would operate roughly as quickly as COPS' non-transactional reads, and the system as a whole would outperform COPS-RT despite offering both read- and write-only transactions and supporting a much more rich data model.

Chapter 8

Related Work

We divide related work into four categories: ALPS systems, causally consistent systems, linearizable systems, and transactional systems. Our work on COPS and Eiger primarily distinguishes itself through scalability—data in a replica (datacenter) may be spread over many machines—and by guaranteeing low latency.

8.1 ALPS Systems

The increasingly crowded category of ALPS systems includes eventually consistent key-value stores such as Amazon's Dynamo [27], LinkedIn's Project Voldemort [63], and the popular memcached [34]. Facebook's Cassandra [46] can be configured to use eventual consistency to achieve ALPS properties, or can sacrifice ALPS properties to provide linearizability. Unlike COPS and Eiger, each of these systems settled for eventual consistency.

A key influence for our work was Yahoo!'s PNUTS [25], which provides per-key sequential consistency (they named this per-record timeline consistency). The PNUTS design does not provide availability, low latency, or partition tolerance because it uses a per-key master to serialize updates. A straight-forward modification that used lamport timestamps for per-key serialization would make it an ALPS system. Per-key sequential consistency, and thus PNUTS, do not provide any consistency between keys, however; achieving such consistency introduces the scalability challenges that COPS addresses.

8.2 Causally Consistent Systems

Many previous system designers have recognized the utility of causal consistency. Bayou [59] provides a SQL-like interface to single-machine replicas that achieves causal+ consistency. Bayou handles all reads and writes locally; it does not address the scalability goals we consider.

TACT [77] is a causal+ consistent system that uses order and numeric bounding to limit the divergence of replicas in the system. The ISIS [15] system exploits the concept of virtual synchrony [14] to provide applications with a causal broadcast primitive (CBcast). CBcast could be used in a straightforward manner to provide a causally consistent key-value store. Replicas that share information via causal memory [2] can also provide a causally consistent ALP key-value store. These systems, however, are not scalable as they all require single-machine replicas.

PRACTI [12] is a causal+ consistent ALP system that supports partial replication, which allows a replica to store only a subset of keys and thus provides some scalability. However, each replica—and thus the set of keys over which causal+ consistency is provided—is still limited to what a single machine can handle.

Lazy replication [45] is closest to the COPS/Eiger approach. Lazy replication explicitly marks updates with their causal dependencies and waits for those dependencies to be satisfied before applying them at a replica. These dependencies are maintained and attached to updates via a front-end that is an analog to our client library. The design of lazy replication, however, assumes that replicas are limited to a single machine: Each replica requires a single point that can (i) create a sequential log of all replica operations, (ii) gossip that log to other replicas, (iii) merge the log of its operations with those of other replicas, and finally (iv) apply these operations in causal order.

Finally, in concurrent theoretical work, Mahajan et al. [55] define real-time causal (RTC) consistency and prove that no consistency strictly stronger than it is achievable in an always-available system. RTC is stronger than causal+ because it enforces a real-time requirement: if causally-concurrent writes do not overlap in real-time, the earlier write may not be ordered after the later write. This real-time requirement helps capture potential causality that is hidden from the system (e.g., out-of-band messaging [24]). In contrast, causal+ does not have a real-time requirement, which allows for more efficient implementations. Notably, COPS's efficient last-writer-wins rule results in a causal+ but not RTC consistent system, while a "return-them-all" conflict handler would provide both properties.

8.3 Linearizable Systems

A large body of research exists about stronger consistency in the wide area. This includes classical research about two-phase commit protocols [66] and distributed consensus (e.g., Paxos [49]). As noted earlier, protocols and systems that provide the strongest forms of consistency—linearizability, sequential consistency, and serializability—are provably incompatible with low latency [10, 52]. Recent examples includes Megastore [11], Spanner [26], and Scatter [38], which use Paxos in the wide-area; CRAQ [70], which is a variant of chain replication [74] that can complete reads in the local-area when there is little write contention, but otherwise requires wide-area operations to ensure linearizability; and Gemini [50], which provides RedBlue consistency with low latency for its blue operations, but high latency for its globally-serialized red operations. In contrast, COPS and Eiger guarantees low latency.

8.4 General Transactions

The database community has long supported consistency across multiple keys through general transactions. In many commercial database systems, a single primary executes transactions across keys, then lazily sends its transaction log to other replicas, potentially over the wide-area. In scale-out designs involving data partitioning (or "sharding"), these transactions are typically limited to keys residing on the same server. Eiger does not have this restriction. More fundamentally, the single primary approach inhibits low-latency, as write operations must be executed in the primary's datacenter.

Several recent systems attempt to reduce the inter-datacenter communication needed to provide transactions. Walter [68] includes optimizations that allow transactions to execute within a single site in many scenarios; nevertheless, ensuring causal relationships between keys can require a two-phase commit across the wide-area. Similarly, MDCC [44] reduces the number of communication rounds involved with transactions for common cases, but at least one round must always traverse the wide-area. Orleans [19], another recent system, also tries to minimize latency for general transaction. Calvin [73] is a recent transaction scheduling and replication layer that can achieve high throughput in the wide-area by replicating inputs instead of effects. These systems all support general transactions, as opposed to Eiger's read- or write-only transactions. They, cannot, however, guarantee low-latency operations, a fundamental goal and property of Eiger. MDCC and Orleans both explicitly acknowledge this by giving programmers the option to receive a fast-but-potentially-incorrect response.

The widely used MVCC algorithm [13, 64] and Eiger maintain multiple versions of objects so they can provide clients with a consistent view of a system. MVCC provides full snapshot isolation, sometimes rejects writes, has state linear in the number of recent reads and writes, and has a sweeping process that removes old versions. Eiger, in contrast, provides only read-only transactions, never rejects writes, has at worst state linear in the number of recent writes, and avoids storing most old versions while using fast timeouts for cleaning the rest.

8.5 Limited Transactions

The implementers of many recent systems have also recognized the utility of limited transactions. Sinfonia [1] provides "mini" transactions to distributed shared memory, but only considers operations within a single datacenter. HBase [40] includes read- and write-only transactions within a single "region," which is a subset of the capacity of a single node.

The designers of Spanner [26], Google's recent linearizable storage system with general transactions and synchronous wide-area replication, also recognized the utility of read-only transactions and included them in their system. Similar to the original distributed read-only transactions [22], Spanner's read-only transactions always take at least two rounds and must block until all involved servers can guarantee they have applied all transactions that committed before the read-only transaction started. In comparison, COPS's and Eiger's read-only transactions normally only take one round and never have to block.

TxCache [61] provides a transactionally consistent, but potential stale, cache for a relational database in a single replica (datacenter) setting. To do this, it tracks validity intervals on versioned data, similar to Eiger's read-only transaction algorithm. TxCache's algorithm is specifically designed to take advantage of the (single) relational database's knowledge of validity intervals. Eiger's algorithm, in contrast, is designed to be entirely distributed with no single point of control.

8.6 Comparing COPS and Eiger

Our work on Eiger [54] builds on the insights we gained and the techniques we developed for COPS [53]. As we show by comparing these systems in Table 8.1, Eiger represents a

	COPS	**COPS-RT**	**Eiger**
Data Model	Key Value	Key Value	**Column Family**
Consistency	**Causal+**	**Causal+**	**Causal+**
Read-Only Txn	No	**Yes**	**Yes**
Write-Only Txn	No	No	**Yes**
Transaction Algos Use	-	Deps	**Logical Time**
Dependencies On	Values	Values	**Operations**
Transmitted Deps	**One-Hop**	All-GarbageC	**One-Hop**
Checked Deps	One-Hop	**Nearest**	One-Hop
Stored Deps	**None**	All-GarbageC	**None**
Garbage Collect Deps	**Unneeded**	Yes	**Unneeded**
Versions Stored	**One**	Few	**Fewer**

Table 8.1: Comparing COPS and Eiger. Entries shown in bold are preferred.

large step forward for causal+ consistent and ALPS systems. In particular, Eiger supports a richer data model, has more powerful transaction support (whose algorithms also work with other consistency models), transmits and stores fewer dependencies, eliminates the need for garbage collection, stores fewer old versions, and is not susceptible to availability problems from metadata explosion when datacenters either fail, are partitioned, or suffer meaningful slow-down for long periods of time.

Chapter 9

Conclusion

Impossibility results divide geo-replicated storage systems into those that can provide the strongest forms of consistency and those that can guarantee low latency. Prior to our work, the general view was that low-latency, scalable distributed storage had to sacrifice strong consistency and settle for eventual consistency, confusing users and programmers. COPS demonstrates that this sacrifice is not fundamental; causal+ consistency is achievable in the ALPS setting. COPS-RT takes this a step farther with its read-only transactions that allow programmers to extract a consistent view of data spread across many machines. Eiger continues to push the envelope on the properties ALPS systems can provide with a richer data model and stronger semantics that include improved read-only transactions and new write-only transactions.

Our major contributions include rigorously defining our setting, designing and implementing scalable systems that provide causal+ consistency, designing and implementing scalable systems that provide stronger semantics, and evaluating our implementations. Rigorously defining our setting includes identifying the ALPS properties, naming causal+ consistency, and formally defining it. Designing and implementing scalable systems that provide causal+ consistency includes our description of COPS, the first scalable causal+ consistent system, and our description of Eiger, the first scalable causal+ consistent system with a rich data model. Designing and implementing stronger semantics for scalable storage includes the non-blocking, lock-free read-only transaction algorithm in COPS-RT; an improved non-blocking, lock-free read-only transaction algorithm in Eiger that is performant, partition tolerant, and applicable to non-causal systems; and a novel write-only transaction algorithm that atomically writes a set of keys, is lock-free, and does not block concurrent read transactions. Evaluating our implementations includes experiments that show COPS has low latency, high throughput, and scales well for all tested workloads; COPS-RT has similar properties for common workloads; and Eiger has performance competitive to eventually-consistent, non-transactional Cassandra.

The key idea in COPS is using explicit dependency metadata and distributed dependency checks to ensure operations appear in the correct causal order without a serialization bottleneck. The key idea in Eiger is that even in a distributed and uncoordinated system, if the data store is updated consistently, there is a consistent result for every query at every logical time. Read-only transactions exploit this by returning results from a single logical time, avoiding the extra dependency tracking necessary in COPS-RT. Write-only transactions exploit this by appearing at a single logical time, even to concurrent read-only transactions.

This leaves applications with two choices for geo-replicated storage. Strongly-consistent storage is required for applications with global invariants, e.g., banking, where accounts cannot drop below zero. COPS-and-Eiger-like systems can serve all other applications, e.g., social networking (Facebook), encyclopedias (Wikipedia), and collaborative filtering (Reddit). These applications no longer need to settle for eventual consistency and can instead make sense of their data with causal+ consistency and stronger semantics.

Bibliography

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 27(3), 2009.

[2] Mustaque Ahamad, Gil Neiger, Prince Kohli, James Burns, and Phil Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, August 2008.

[4] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Conf. Software Engineering*, October 1976.

[5] Simple storage service. `http://aws.amazon.com/s3/`, 2012.

[6] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, October 2009.

[7] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[8] `http://thrift.apache.org/`, 2011.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[10] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, January 2011.

[12] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *NSDI*, May 2006.

[13] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computer Surveys*, 13(2), June 1981.

[14] Kenneth P. Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, November 1987.

[15] Kenneth P. Birman and Robbert V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[16] W.J. Bolosky, D. Bradshaw, R.B. Haagens, N.P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, April 2011.

[17] Eric Brewer. Towards robust distributed systems. PODC Keynote, July 2000.

[18] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *PPoPP*, Bangalore, India, January 2010.

[19] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *SOCC*, 2011.

[20] B Calder, J Wang, A Ogus, N Nilakantan, A Skjolsvold, S McKelvie, Y Xu, S Srivastav, J Wu, H Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[21] `http://cassandra.apache.org/`, 2012.

[22] Arvola Chan and Robert Gray. Implementing distributed read-only transactions. *IEEE Trans. Info. Theory*, 11(2), 1985.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2), 2008.

[24] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, December 1993.

[25] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, August 2008.

[26] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, October 2012.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, October 2007.

[28] Phil Dixon. Shopzilla site redesign: We get what we measure. Velocity Conference Talk, 2009.

[29] eBay. Personal communication, 2012.

[30] Facebook. Personal communication, 2011.

[31] https://github.com/vrv/FAWN-KV, 2011.

[32] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[33] Jack Ferris. The TAO graph database. CMU PDL Talk, April 2012.

[34] Brad Fitzpatrick. Memcached: a distributed memory object caching system. http://memcached.org/, 2011.

[35] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *SOSP*, October 1997.

[36] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, October 2003.

[37] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[38] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *SOSP*, Cascais, Portugal, October 2011.

[39] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, August 2009.

[40] http://hbase.apache.org/, 2012.

[41] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[42] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[43] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM TOCS*, 10(3), February 1992.

[44] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.

[45] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[46] Avinash Lakshman and Prashant Malik. Cassandra – a decentralized structured storage system. In *LADIS*, October 2009.

[47] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[48] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computer*, 28(9), 1979.

[49] Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[50] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, October 2012.

[51] Greg Linden. Make data useful. Stanford CS345 Talk, 2006.

[52] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[53] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, October 2011.

[54] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, April 2013.

[55] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report TR-11-22, Univ. Texas at Austin, Dept. Comp. Sci., 2011.

[56] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM TOCS*, 29(4), 2011.

[57] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM TOPLAS*, 8(1), January 1986.

[58] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3), 1983.

[59] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, October 1997.

[60] Larry Peterson, Andy Bavier, and Sapan Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci., 2011.

[61] Dan R.K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, October 2010.

[62] http://www.nmc-probe.org/, 2013.

[63] http://project-voldemort.com/, 2011.

[64] David P. Reed. *Naming and Synchronization in a Decentralized Computer Systems*. PhD thesis, Mass. Inst. of Tech., 1978.

[65] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. Velocity Conference Talk, 2009.

[66] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Info. Theory*, 9(3), May 1983.

[67] http://code.google.com/p/snappy/, 2011.

[68] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP*, October 2011.

[69] A read-optimized globally distributed store for social graph data. Under Submission, 2013.

[70] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX ATC*, June 2009.

[71] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Conf. Parallel Distributed Info. Sys.*, September 1994.

[72] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Sys.*, 4(2), 1979.

[73] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, May 2012.

[74] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, December 2004.

[75] http://vicci.org/, 2012.

[76] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, December 2002.

[77] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, October 2000.

Appendix A

Formal Definition of Causal+

We first present causal consistency with convergent conflict handling (causal+ consistency) for a system with only read and write operations, and we then introduce read-only transactions. We use a model closely derived from Ahamad et al. [2], which in turn was derived from those used by Herlihy and Wing [41] and Misra [57].

Original Model of Causal Consistency [2] with terminology modified to match this paper's definitions:

A system is a finite set of threads of execution, also called threads, that interact via a key-value store that consists of a finite set of keys. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of threads. The local history L_i of a thread i is a sequence of read and write operations. If operation σ_1 precedes σ_2 in L_i, we write $\sigma_1 \underset{i}{\rightarrow} \sigma_2$. A history $H = \langle L_1, L_2, \ldots, L_n \rangle$ is the collection of local histories for all threads of execution. A serialization S of H is a linear sequence of all operations in H in which each get on a key returns its most recent preceding write (or $\perp$ if there does not exist any preceding write). The serialization S respects an order $\rightarrow$ if, for any operation σ_1 and σ_2 in S, $\sigma_1 \rightarrow \sigma_2$ implies σ_1 precedes σ_2 in S.

The *writes-into* order associates a write operation, `write`(k,v), with each read operation, `read`(k)=v. Because there may be multiple writes of a value to a key, there may be more than one writes-into order.[1] A writes-into order $\mapsto$ on H is any relation with the following properties:

- If $\sigma_1 \mapsto \sigma_2$, then there is a key k and value v such that operation σ_1 := `write`(k,v) and σ_2 := `read`(k)=v.
- For any operation σ_2, there exists at most one σ_1 for which $\sigma_1 \mapsto \sigma_2$.
- If σ_2 := `read`(k)=v for some k,v and there exists no σ_1 such that $\sigma_1 \mapsto \sigma_2$, then v = $\perp$. That is, a read with no preceding write must retrieve the initial value.

Two operations, σ_1 and σ_2, are related by a causal order $\leadsto$ if and only if one of the following holds:

[1] The COPS and Eiger systems uniquely identifies values with timestamps so there is only one writes-into order, but this is not necessarily true for causal+ consistency in general.

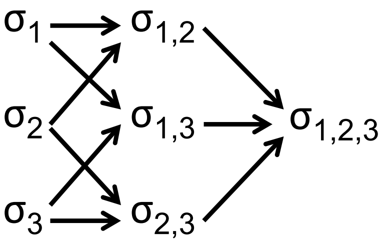
- $\sigma_1 \xrightarrow[i]{} \sigma_2$ for some t_i (σ_1 precedes σ_2 in L_i);
- $\sigma_1 \mapsto \sigma_2$ (σ_2 reads the value written by σ_1); or
- There is some other operation σ' such that $\sigma_1 \rightsquigarrow \sigma' \rightsquigarrow \sigma_2$.

Incorporating Convergent Conflict Handling. Two operations on the same key, σ_1:=`write`(k,v$_1$) and σ_2:=`write`(k,v$_2$), are in *conflict* if they are not related by causality: $\sigma_1 \not\rightsquigarrow \sigma_2$ and $\sigma_2 \not\rightsquigarrow \sigma_1$.

A *convergent conflict handling function* is an associative, commutative function that operates on a set of conflicting operations on a key to eventually produce one (possibly new) final value for that key. The function must produce the same final value independent of the order in which it observes the conflicting updates. In this way, once every replica has observed the conflicting updates for a key, they will all independently agree on the same final value.

We model convergent conflict handling as a set of *handler* threads that are distinct from normal client threads. The handlers operate on a pair of conflicting values (v_1, v_2) to produce a new value $newval = h(v_1, v_2)$. By commutativity, $h(v_1, v_2) = h(v_2, v_1)$. To produce the new value, the handler thread had to read both v_1 and v_2 before writting the new value, and so *newval* is causally ordered after both original values: $v_1 \rightsquigarrow newval$ and $v_2 \rightsquigarrow newval$.

With more than two conflicting updates, there will be multiple invocations of handler threads. For three values, there are several possible orders for resolving the conflicting updates in pairs:



The commutativity and associativity of the handler function ensures that regardless of the order, the final output will be identical. Further, it will be causally ordered after all of the original conflicting writes, as well as any intermediate values generated by the application of the handler function. If the handler observes multiple pairs of conflicting updates that produce the same output value (e.g., the final output in the figure above), it must output only one value, not multiple instances of the same value.

To prevent a client from seeing and having to reason about multiple, conflicting values, we restrict the write set for each **client** thread to be conflict free, denoted p_{cf_i}. A write set is *conflict free* if $\forall \sigma_j, \sigma_k \in p_{cf_i}$, σ_j and σ_k are not in conflict; that is, either they are writes to different keys or causally-related writes to the same key. For example, in the three conflicting write example, p_{cf_i} might include σ_1, $\sigma_{1,2}$, and $\sigma_{1,2,3}$, but not σ_2, σ_3, $\sigma_{1,3}$, or $\sigma_{2,3}$. The conflict-free property applies to client threads and not handler threads purposefully. Handler threads must be able to read values from conflicting writes so they may reason about and resolve them; client threads should not see conflicts so they do not have to reason about them.

Adding handler threads models the new functionality that convergent conflict handling provides. Restricting the write set strengthens consistency from causal to causal+. There are causal executions that are not causal+: for example, if σ_1 and σ_2 conflict, a client may read the value write by σ_1 and then the value written by σ_2 in a causal, but not causal+, system. On the other hand, there are no causal+ executions that are not causal, because causal+ only introduces an additional restriction (a smaller write set) to causal consistency.

If H is a history and t_i is a thread, let $A^H_{i+p_{cf_i}}$ comprise all operations in the local history of t_i, and a conflict-free set of writes in H, p_{cf_i}. A history H is *causally consistent with convergent conflict handling* (causal+) if it has a causal order $\rightsquigarrow$, such that

Causal+: For each *client* thread of execution t_i, there is a serialization S_i of $A^H_{i+p_{cf_i}}$ that respects $\rightsquigarrow$.

A data store is *causal+ consistent* if it admits only causal+ histories.

Introducing Read-only Transactions. To add read-only transactions to the model, we redefine the writes-into order so that it associates N write operations, `write`(k,v), with each read-only transaction of N values, `read_trans`($[k_1,\ldots,k_N]$)=$[v_1,\ldots,v_N]$. Now, a writes-into order $\mapsto$ on H is any relation with the following properties:

- If $\sigma_1 \mapsto \sigma_2$, then there is a k and v such that σ_1 := `write`(k,v) and σ_2 := `read_trans`([…,k,…])=[…,v,…]. That is, for each component of a read-only transaction, there exists a preceding write.
- For each component of a read-only transaction σ_2, there exists at most one σ_1 for which $\sigma_1 \mapsto \sigma_2$.
- If σ_2 := `read_trans`([…,k,…])=[…,v,…] for some k,v and there exists no σ_1 such that $\sigma_1 \mapsto \sigma_2$, then v=$\perp$. That is, a read with no preceding write must retrieve the initial value.